

# Softwaretechnik Zusammenfassung

Maximilian Wolf

23. Februar 2026

# Inhaltsverzeichnis

<b>1 Folie 1: Introduction</b>	<b>7</b>
<b>2 Folie 2: Requirements</b>	<b>7</b>
2.1 Requirements Engineering . . . . .	8
2.2 Dokumentation . . . . .	9
2.2.1 Natural Language Specification . . . . .	9
2.2.2 Structured Specification . . . . .	10
2.2.3 Use Case Diagrams (Anwendungsfalldiagramm) . . . . .	10
<b>3 Folie 3: System Modeling</b>	<b>11</b>
3.1 Unified Modeling Language (UML) . . . . .	11
3.1.1 Structure Diagrams (Strukturdiagramme) . . . . .	11
3.1.2 Behavior Diagrams (Verhaltensdiagramme) . . . . .	11
3.2 Domain Models . . . . .	12
3.2.1 Inhalt . . . . .	12
3.3 Activity Diagrams (Aktivitätsdiagramme) . . . . .	12
3.3.1 Grundelemente . . . . .	13
3.3.2 Steuerungselemente . . . . .	13
3.3.3 Verantwortlichkeiten (Swimlanes) . . . . .	13
3.4 State Machine Diagrams (Zustandsdiagramme) . . . . .	13
3.4.1 Kernkonzepte . . . . .	13
3.4.2 Hierarchische Zustände . . . . .	14
<b>4 Folie 4: Software Architecture</b>	<b>14</b>
4.1 Goals of Software Architecture . . . . .	14
4.2 Architectural Views . . . . .	14
4.2.1 Logical View . . . . .	14
4.2.2 Process View . . . . .	15
4.2.3 Development View . . . . .	15
4.2.4 Physical View . . . . .	15
4.3 Component Diagrams . . . . .	16
4.4 Architekturmuster . . . . .	17
4.4.1 Layered Architecture . . . . .	17
4.4.2 Client-Server Architecture (2-Layer Architecture) . . . . .	17
4.4.3 3-Tier Architecture (3-Layer Architecture) . . . . .	17
4.4.4 Peer-to-Peer Architecture . . . . .	17
4.4.5 Model-View-Controller (MVC) Architecture . . . . .	18
4.4.6 Pipe-and-Filter Architecture . . . . .	18

<b>5 Folie 5: Software Design</b>	<b>18</b>
5.1 Classdiagrams (Klassendiagramme) . . . . .	18
5.1.1 Attribute und Operationen . . . . .	19
5.2 Aggregation und Komposition . . . . .	19
5.3 Inheritance . . . . .	20
5.3.1 Generalisierung . . . . .	20
5.3.2 Abstrakte Klasse . . . . .	20
5.4 Sequence Diagrams (Sequenzdiagramme) . . . . .	21
<b>6 Folie 6: Implementation</b>	<b>22</b>
6.1 Code Formatierung . . . . .	22
6.2 Namenskonventionen (Rules of Naming) . . . . .	22
6.3 Kommentare (Code Dokumentation) . . . . .	23
6.4 Werkzeuge und Umgebung (Tools and Environments) . . . . .	23
6.4.1 Definitionen . . . . .	23
6.4.2 Übersicht gängiger Development Tools . . . . .	23
<b>7 Folie 7: Design Patterns</b>	<b>24</b>
7.1 Strukturmuster . . . . .	25
7.1.1 Object Adapter Pattern . . . . .	25
7.1.2 Composite Pattern . . . . .	26
7.1.3 Decorator Pattern . . . . .	26
7.2 Erzeugungsmuster . . . . .	28
7.2.1 Singleton Pattern . . . . .	28
7.2.2 Abstract Factory Pattern (aka: Kit) . . . . .	28
7.3 Verhaltensmuster . . . . .	29
7.3.1 Observer Pattern . . . . .	30
<b>8 Folie 8: Quality Assurance</b>	<b>31</b>
8.1 Expectation on Quality . . . . .	32
8.2 Product Quality . . . . .	33
8.3 Quality in Use . . . . .	33
8.4 Software Testing . . . . .	34
8.5 Stages of Testing . . . . .	34
8.6 Code-Reviews . . . . .	35
8.7 White-Box-Testing (Strukturtests) . . . . .	35
8.7.1 Testfälle . . . . .	35
8.7.2 Test-case Design (Testfallentwurf) . . . . .	36
8.7.3 White-Box-Testing (Strukturtest) . . . . .	37
8.7.4 Coverage Criteria (Überdeckungskriterien) . . . . .	38
8.8 Black-Box-Tests Testing . . . . .	39

8.8.1	Äquivalenzklassentests . . . . .	39
8.8.2	Grenzwerttests (Boundary Testing) . . . . .	40
8.8.3	Gründe für positive Testfälle (Fehlerfunde) . . . . .	40
<b>9</b>	<b>Folie 9: Development Process</b>	<b>41</b>
9.1	Das Wasserfallmodell (The Waterfall Model) . . . . .	41
9.1.1	Die 5 Phasen . . . . .	41
9.2	Das V-Modell (V-Model for Extensive Testing) . . . . .	42
9.2.1	Die 4 Teststufen . . . . .	42
9.3	Motivation . . . . .	43
9.3.1	Agile (Development) Methods . . . . .	44
9.3.2	Agile Manifesto . . . . .	44
9.3.3	Scrum . . . . .	45
9.3.4	Vor- und Nachteile . . . . .	47
<b>10</b>	<b>Folie 10: Project Management</b>	<b>47</b>
10.1	Definition und Ziele . . . . .	47
10.2	Risikomanagement . . . . .	48
10.2.1	Risikoklassifikation . . . . .	48
10.2.2	Risikostrategien und Phasen . . . . .	48
10.3	Personalmanagement . . . . .	49
10.4	Projekt- und Terminplanung . . . . .	50
10.4.1	Visualisierungstools . . . . .	50
10.4.2	Konstruktion eines Netzplans (Network Diagram) . . . . .	50
<b>11</b>	<b>Folie 11: Configuration Management</b>	<b>52</b>
11.1	Konfigurationsmanagement und Versionskontrolle . . . . .	52
11.1.1	Vier Aktivitäten im Konfigurationsmanagement . . . . .	52
11.2	Entwicklungsphasen . . . . .	52
11.3	Versionskontrolle . . . . .	53
11.3.1	Ziele der Versionskontrolle . . . . .	53
11.3.2	Version Control Systems (VCS) . . . . .	53
11.3.3	Terminologie der Versionskontrolle . . . . .	53
11.3.4	Operationen von Version Control Systems . . . . .	54
11.3.5	Gute Commit-Nachrichten . . . . .	54
11.3.6	Merging . . . . .	55
11.4	Systembau (System Building) . . . . .	55
11.4.1	Build-Umgebungen . . . . .	55
11.4.2	Werkzeuge für Build und Systemintegration . . . . .	55
11.5	Continuous Integration (CI) und Deployment (CD) . . . . .	56
11.5.1	Schritte in der kontinuierlichen Integration . . . . .	56

<b>12 Folie 12: Software Evolution</b>	<b>57</b>
12.1 Lehman's Laws of Software Evolution . . . . .	57
12.2 Bedeutung von Software Evolution . . . . .	57
12.3 Software Evolution Faktoren . . . . .	58
12.4 Arten von Veränderungen . . . . .	58
12.5 Code Refactoring . . . . .	58
12.6 Code Smells . . . . .	58
12.6.1 Refactoring Methoden . . . . .	58
12.6.2 Beispiele . . . . .	59
12.7 Zusammenfassung . . . . .	59
<b>13 Folie 13: Software Maintenance</b>	<b>60</b>
13.1 Arten von Maintenance . . . . .	60
13.2 Maintenance vs. Evolution . . . . .	60
13.2.1 Wartung . . . . .	61
13.2.2 Evolution . . . . .	61
13.3 Reengineering . . . . .	61
13.3.1 Reverse Engineering . . . . .	61
13.3.2 Forward Engeneering . . . . .	61
13.3.3 Transformative Engineering / Restructuring . . . . .	62
13.3.4 Reengineering . . . . .	62
13.3.5 Legacy Systems . . . . .	62
13.4 Migration . . . . .	63
13.4.1 Migrationsstrategien . . . . .	63
13.5 Zusammenfassung . . . . .	63
<b>14 Folie 14: Compilation and Static Analysis</b>	<b>64</b>
14.1 Grundlagen der Kompilierung . . . . .	64
14.1.1 Programmiersprachen . . . . .	64
14.1.2 Compilation vs. Interpretation . . . . .	64
14.1.3 Intermediate Languages . . . . .	64
14.1.4 Compiler Optimierung . . . . .	65
14.1.5 Nicht-korrektter Code . . . . .	66
14.1.6 Compiler Architecture . . . . .	66
14.1.7 Typensicherheit und Korrektheit . . . . .	67
14.1.8 Klassifizierung von Fehlerarten . . . . .	67
14.2 Missverständnisse bezüglich Performance . . . . .	68
14.2.1 Compiler-Optimierungen . . . . .	68
14.2.2 Missverständnis 1: Arrayzugriffe . . . . .	68
14.2.3 Missverständnis 2: Schleifen . . . . .	69
14.2.4 Missverständnis 3: Methodenaufrufe . . . . .	70

14.2.5	Missverständnis 4: Objekte . . . . .	70
14.2.6	Missverständnis 5: Garbage Collection . . . . .	71
14.3	Test-Driven Entwicklung und Design per Vertrag . . . . .	71
14.3.1	Fehlerarten . . . . .	71
14.3.2	Design by Contract (Methodenverträge) . . . . .	71
14.3.3	Generated Assertions und Blame Assignment . . . . .	72
14.3.4	Behavioral Subtyping . . . . .	72
<b>15 Folie 15: Software Product Lines</b>		<b>72</b>
15.1	Software Product Lines and Feature Modeling . . . . .	72
15.1.1	Grundlagen . . . . .	72
15.1.2	Feature Modeling . . . . .	73
15.1.3	Valid Configurations . . . . .	74
15.2	Implementierung von Software Product Lines . . . . .	74
15.2.1	Strategien . . . . .	74
15.3	Testen von Software Product Lines . . . . .	75
15.3.1	Test-Ansätze . . . . .	75
15.3.2	Feature Interaktionen . . . . .	75
15.3.3	Pairwise Interaction Testing (PIT) . . . . .	75
15.3.4	Zusammenfassung und Trade-Off . . . . .	76
<b>16 Folie 16: Open-Source Software</b>		<b>76</b>
16.1	Open-Source Entwicklung . . . . .	76
16.1.1	Kriterien für Verwendung von open-source Komponenten	76
16.1.2	Richard Stallman's Four Freedoms . . . . .	77
16.1.3	Free Software . . . . .	77
16.2	Open-Source Principles . . . . .	77
16.2.1	Copyleft und Copyright . . . . .	78
16.3	Open-Source Lizenzen . . . . .	78
16.3.1	MIT License . . . . .	78
16.3.2	GPL: GNU General Public License . . . . .	79
16.3.3	LGPL: GNU Lesser General Public License . . . . .	79
16.3.4	Re-Licensing und Dual-Licensing . . . . .	79
<b>17 Folie 17: Automotive Software Architectures</b>		<b>80</b>
17.1	Layered Architecture . . . . .	80
17.2	Pipe-and-Filter Architecture . . . . .	80
17.3	Client-Server Architecture . . . . .	81
17.4	Publish-Subscribe Architecture . . . . .	81
17.5	Centralized Software Architecture . . . . .	82

# 1 Folie 1: Introduction

## Definition 1.1: Software

Computer Programme und alle assoziierten Dokumentationen, Libraries, Webseiten und Konfigurations Daten.

- Application: Anwendungssoftware / Anwendung
- System Software: Plattform für andere Software

# 2 Folie 2: Requirements

- **User Requirements** (Benutzeranforderungen): Anforderungen in natürlicher Sprache an Services, die das System für Benutzer bereitstellen soll und die Einschränkungen, unter welchen das System arbeiten soll
- **System Requirements** (Systemanforderungen): Detaillierte Beschreibungen der Funktionen, Services und betriebliche Einschränkungen des Systems und die Komponenten, die implementiert werden sollen
- **Functional Requirements**: Aussagen über Services, die das System bereitstellt, wie das System auf bestimmte Eingaben und in bestimmten Situationen reagiert
- **Non-Functional Requirements**: Einschränkungen der Services bzw. Funktionen eines Systems z.B. Anforderungen für Latenzen und Standards, Einschränkungen an dem Entwicklungsprozess

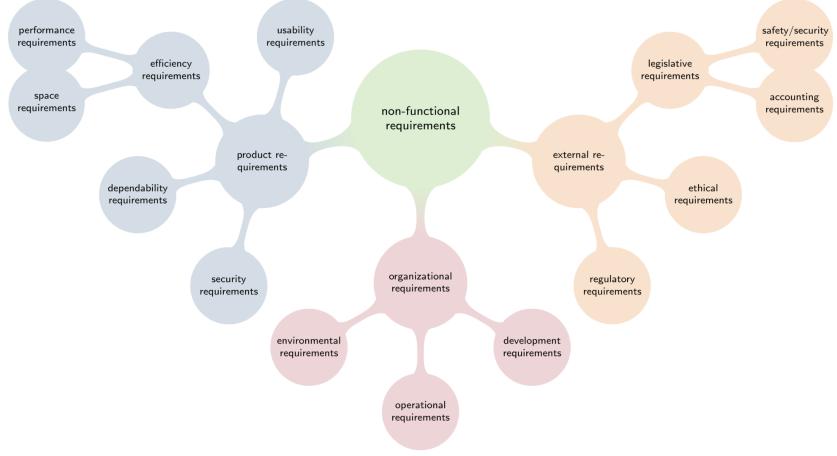


Abbildung 1: Functional Requirements

## 2.1 Requirements Engineering

- **Requirements Elicitation and Analysis:** Herleitung der Systemsanforderungen durch betrachten des existierenden Systems bzw. Entwicklung von Prototypen
- **Requirements Specification:** Notieren der Anforderungen eines Systems in einem Anforderungs-Dokument
- **Requirements Validation:** Überprüfung der Anforderungen auf Umsetzbarkeit und Ganzheit
  - **Validity Checks:** Spiegeln Anforderungen echte Bedürfnisse wieder?
  - **Consistency Checks:** Gibt es überflüssige / sich widersprechende Anforderungen?
  - **Completeness Checks:** Sind alle Funktionen und Einschränkungen dokumentiert?
  - **Realism Checks:** Sind die Anforderungen umsetzbar?
  - **Verifiability:** Lassen sich die Anforderungen überprüfen?

## 2.2 Dokumentation

### 2.2.1 Natural Language Specification

Regel	Kurzbezeichnung	Beschreibung
R1	Aktive Stimme	Jede Anforderung muss in der aktiven Stimme formuliert werden. Der Akteur (System oder Benutzer) muss klar spezifiziert sein.
R2	Volle Verben	Prozesse müssen durch vollständige Verben ausgedrückt werden (z.B. „liest“, „generiert“), nicht durch unvollständige Hilfsverben wie „ist“ oder „hat“.
R3	Unvollständige Prozesse	Unvollständig spezifizierte Prozesswörter (Verben) müssen durch das Hinzufügen fehlender Begriffe (Objekte/Komplemente) zu vollständigen Aussagen ergänzt werden.
R4	Unvollständige Bedingungen	Bei Wenn-Dann-Sonst-Bedingungen müssen sowohl der „Dann“- als auch der „Sonst“-Fall klar beschrieben werden.
R5	Universelle Quantoren	Aussagen mit „nie“, „immer“, „jeder“, „kein“ oder „alle“ müssen auf universelle Gültigkeit geprüft und gegebenenfalls eingeschränkt werden.
R6	Nominalstrukturen	Nomen (z.B. „Registrierung“) deuten oft auf einen komplexen Prozess hin, der durch die Verwendung von Verben präziser beschrieben werden sollte.
R7	Indefinite Nomen	Unklare Nomen (z.B. „Benutzer“, „Nachricht“) müssen präzisiert werden, um klarzustellen, ob ein generischer Begriff oder ein konkretes Objekt gemeint ist.
R8	Verantwortlichkeiten	Wenn etwas möglich, unmöglich, erlaubt oder notwendig ist, muss geklärt werden, wer (oder was) dies durchsetzt oder verhindert.
R9	Implizite Annahmen	Begriffe in den Anforderungen, die nicht erklärt werden, deuten oft auf implizierte Annahmen hin, die explizit spezifiziert werden müssen.

Tabelle 1: Regeln zur Formulierung von Anforderungen

### 2.2.2 Structured Specification

- Verwendung von Templates
- Unterscheidung zwischen **Function**, **Description**, **Precondition**, **Post-condition** und **Rationale**

### 2.2.3 Use Case Diagrams (Anwendungsfalldiagramm)

- Visualisierung der Anforderungen an ein System
- **Subjekt**: Jedes Subjekt stellt ein in Betracht gezogenes System dar, für das der Anwendungsfall gilt
- **Akteur**: Jeder Benutzer und jedes andere System, das mit einem Subjekt interagieren kann, wird als Akteur repräsentiert
- **Anwendungsfall**: Spezifikation des Verhaltens in Form von Verb und Nomen
- **Beziehungen**
  - **Include**: inkludierter Anwendungsfall wird immer auch ausgeführt, wenn der Basis Anwendungsfall (base use case) ausgeführt wird
  - **Extend**: wenn der Basis Anwendungsfall ausgeführt wird, wird der Anwendungsfall vielleicht auch ausgeführt

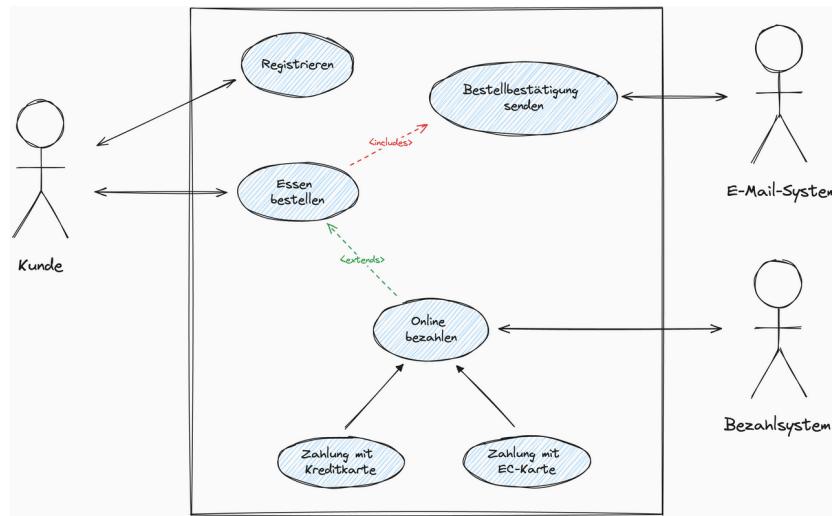


Abbildung 2: Use Case Diagram

### 3 Folie 3: System Modeling

#### 3.1 Unified Modeling Language (UML)

##### 3.1.1 Structure Diagrams (Strukturdiagramme)

bilden das statische Gerüst eines Systems ab

- **Fokus:** zeigen statische Struktur der Objekte in einem System.
- **Zeit:** Die dargestellten Elemente sind zeitunabhängig (irrespective of time).
- **Inhalt:** repräsentieren bedeutungsvolle Konzepte der Anwendung, wie zum Beispiel abstrakte Konzepte, reale Objekte oder Implementierungsdetails.
- **Beispiele:** Klassendiagramm (Class Diagram), Objektdiagramm, Komponentendiagramm.

##### 3.1.2 Behavior Diagrams (Verhaltensdiagramme)

beschreiben Dynamik und Abläufe innerhalb des Systems

**Fokus:** zeigen das dynamische Verhalten der Systemobjekte.

**Zeit:** Verhalten wird als Serie von Änderungen am System über die Zeit definiert.

**Inhalt:** Sie visualisieren Methoden, Kollaborationen, Aktivitäten und Zustandsverläufe.

**Beispiele:** Aktivitätsdiagramm (Activity Diagram), Zustandsdiagramm (State Machine Diagram)

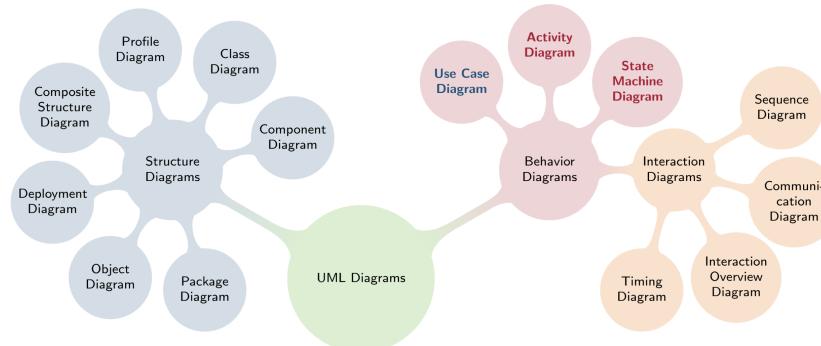


Abbildung 3: UML Diagrams

## 3.2 Domain Models

dient dazu, die relevanten Elemente eines Anwendungsbereichs und deren Beziehungen abzubilden

- **Zweck:** Abbildung der fachlichen Begriffe und Zusammenhänge der Anwendungsdomäne zur Unterstützung des gemeinsamen Verständnisses.
- **Notation:** statische Struktur in Form eines UML-Klassendiagramms.

### 3.2.1 Inhalt

- **Klassen:** Objekte mit gleichen Attributen und Semantik
- **Assoziationen:** strukturelle Beziehungen zwischen Objekten inklusive Multiplizitäten
- **Rollen:** beschreiben dabei die spezifische Funktion einer Klasse innerhalb einer Beziehung.
- **Keine** Modellierung von Methoden, Implementierungsdetails

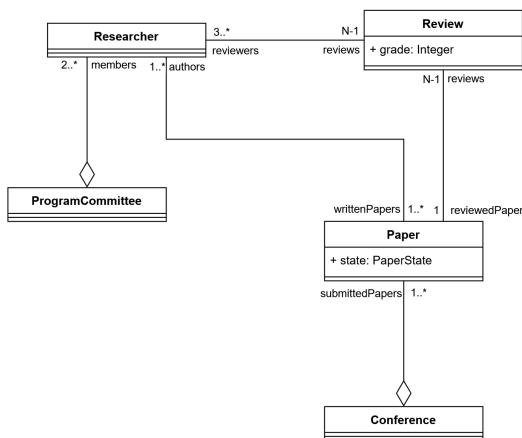


Abbildung 4: Domain Model

## 3.3 Activity Diagrams (Aktivitätsdiagramme)

Visualisierung von Aktivitäten und deren Ausführungsreihenfolge (Kontrollfluss)

### 3.3.1 Grundelemente

- **Aktivitäten:** Abgerundete Boxen, die Aufgaben repräsentieren.
- **Flüsse (Flows):** Durchgezogene Pfeile, welche die Reihenfolge vorgeben.
- **Start/Ende:** gefüllter Kreis markiert den Beginn, ein "Bull's Eye" den Abschluss.

### 3.3.2 Steuerungselemente

- **Verzweigung (Branching):** Raute mit einem Eingang und mehreren Ausgängen, die durch Guards (Bedingungen in eckigen Klammern) gesteuert werden.
- **Zusammenführung (Merging):** Führt mehrere alternative Pfade wieder in einen Fluss zusammen.
- **Gabelung (Forking):** dicker Balken, der einen Fluss in mehrere parallele/gleichzeitige Aktivitäten aufteilt.
- **Vereinigung (Joining):** Führt parallele Flüsse wieder zu einem einzelnen Fluss zusammen.

### 3.3.3 Verantwortlichkeiten (Swimlanes)

Rechteckige Bereiche, die Aktivitäten nach Verantwortlichkeiten gruppieren (z.B. "Student" vs. "Supervisor").

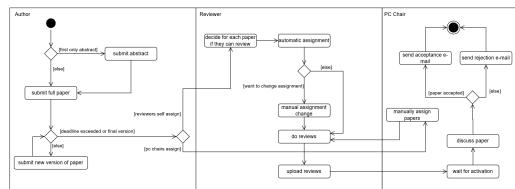


Abbildung 5: Activity Diagram

## 3.4 State Machine Diagrams (Zustandsdiagramme)

### 3.4.1 Kernkonzepte

- **Zustand (State):** Rechteck mit abgerundeten Ecken und durch bestimmte Bedingung oder Situation charakterisiert.

- **Ereignis** (Event): löst einen Zustandsübergang aus.
- **Übergang** (Transition): durch Pfeil dargestellte Beziehung zwischen zwei Zuständen.
- **Start-/ Endzustand**: genau ein initialer Zustand (gefüllter Kreis) und ein finaler Zustand (Bullseye).

### 3.4.2 Hierarchische Zustände

- **Einfacher Zustand** (Simple State): Zustand ohne weitere Unterstruktur.
- **Komplexer Zustand** (Composite State): Zustand, der verschachtelte Unterzustände (Substates) enthält.
- Jeder komplexe Zustand muss seinen eigenen initialen Startzustand besitzen.

## 4 Folie 4: Software Architecture

### Definition 4.1: Architekturentwurf

Organisation eines Systems, sodass es die Funktionalen und Nicht-Funktionalen Anforderungen erfüllt

### 4.1 Goals of Software Architecture

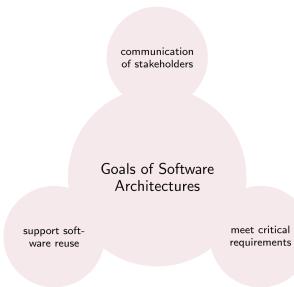


Abbildung 6: Goals of Software Architecture

### 4.2 Architectural Views

#### 4.2.1 Logical View

- **Inhalt**: Funktionalität, die das System für die Endbenutzer bereitstellt.

- **UML-Umsetzung:** primär durch Domänenmodell (als Klassendiagramm).
- **Ziel:** Unterstützung des gegenseitigen Verständnisses der Problemdomäne.

#### 4.2.2 Process View

- **Inhalt:** beschreibt die dynamischen Aspekte des Systems, also Abläufe und das Verhalten über die Zeit.
- **UML-Umsetzung:** Verhaltensdiagramme (Behavior Diagrams):
  - **Aktivitätsdiagramme:** Fokus auf den Kontrollfluss von Aktivität zu Aktivität.
  - **Zustandsdiagramme:** Fokus auf Zustände und Übergänge als Reaktion auf Ereignisse

#### 4.2.3 Development View

- **Inhalt:** illustriert System aus der Perspektive der Programmierer und des Software-Managements.
- **UML-Umsetzung:** Strukturdiagramme wie das Komponentendiagramm oder detaillierte Klassendiagramme, die über das Domänenmodell hinausgehen und Implementierungskonzepte enthalten.
- **Ziel:** Beschreibung des Systems für die Ingenieure, die es implementieren.

#### 4.2.4 Physical View

- **Inhalt:** zeigt physische Anordnung der Softwarekomponenten auf der Hardware (Deployment).
- **UML-Umsetzung:** Deployment Diagram (Verteilungsdiagramm) aufgeführt.
- **Ziel:** Visualisierung der Verteilung der Software-Artefakte auf die physische Infrastruktur.

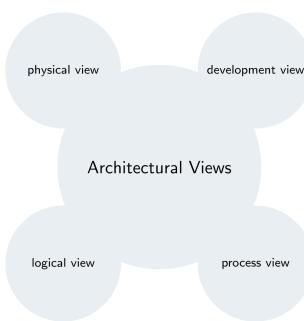


Abbildung 7: Architectural Views

### 4.3 Component Diagrams

**Definition 4.2: Komponente**

Austauschbarer Teil eines Systems, stellt eine Implementierung einer bzw. mehrerer Interfaces dar

**Definition 4.3: Interface**

Spezifikation einer Sammlung von Operationen, die eine bereitstellen haben muss

Darstellung der physischen oder logischen Strukturierung eines Systems in modulare Bausteine.

- **Komponente:** Eine kapselbare, modulare Einheit des Systems, die eine bestimmte Funktionalität bereitstellt und als ersetzbarer Baustein dient.
- **Angebotene Schnittstelle (Provided Interface):** Definiert Dienste, welche die Komponente für andere Systemteile bereitstellt (Notation: Kreis bzw. „Lollipop“).
- **Benötigte Schnittstelle (Required Interface):** Definiert Dienste, welche die Komponente von anderen Bausteinen benötigt, um ihre Aufgabe zu erfüllen (Notation: Halbkreis bzw. „Socket“).
- **Abhängigkeiten (Dependencies):** Zeigen durch gerichtete Pfeile an, dass eine Komponente Informationen oder Dienste einer anderen Komponente oder Schnittstelle nutzt.
- **Ports:** Spezifische Interaktionspunkte auf der Hülle einer Komponente, die den internen Aufbau von der äußeren Umgebung isolieren.

## 4.4 Architekturmuster

### 4.4.1 Layered Architecture

- Verwaltung von Subsystemen ist komplex, daher werden diese in Schichten unterteilt
- Untere Schichten stellen Services für obere Schichten dar, obere Schichten geben Aufgaben an untere Schichten weiter
- **Layer-Arten**
  - **Strict Layers**: jede Schicht kann nur die auf die nächste zugreifen
  - **Relaxed Layers**: jede Schicht kann auf alle darunter liegenden zugreifen

### 4.4.2 Client-Server Architecture (2-Layer Architecture)

- Mehrere Clients müssen auf gleiche Daten zugreifen
- Aufteilung in **Server** und **Clients**
- Clients initiieren Verbindung mit Server

### 4.4.3 3-Tier Architecture (3-Layer Architecture)

- Clients mit gleicher Funktionalität aber unterschiedlicher Darstellung erforderlich
- Aufteilung zwischen **data presentation**, **application logic** und **data management**
- **Client-Typen**
  - **Thin-Client Application**: Anwendungs-Logik serverseitig
  - **Thick-Client Application**: Anwendungs-Logik clientseitig

### 4.4.4 Peer-to-Peer Architecture

- **Problem**: hohe Last auf Server und Risiko für Versagen des Servers
- **Idee**: dezentralisiertes Übertragen von Daten
- Peers übernehmen Client- und Serverrolle, verbinden sich miteinander und tauschen Daten direkt aus
- Arbiträre und dynamische **Topologie**

#### 4.4.5 Model-View-Controller (MVC) Architecture

- **Model:** repräsentiert die Daten und die Geschäftslogik, entspricht dem Domänenmodell, das die fachlichen Konzepte und deren statische Beziehungen abbildet.
- **View:** Darstellung der Daten für den Benutzer, Verwendung mehrerer UML-Diagramme, um unterschiedliche Sichten (Perspektiven) auf das System zu ermöglichen.
- **Controller:** vermittelt zwischen Model und View, wird häufig durch Verhaltensdiagramme (Aktivitäts- und Zustandsdiagramme) modelliert.

#### 4.4.6 Pipe-and-Filter Architecture

beschreibt Systeme, in denen Datenströme nacheinander von verschiedenen Komponenten (Filtern) verarbeitet werden:

- **Konzept:** Daten fließen durch eine "Pipe" von einer Verarbeitungseinheit zur nächsten.
- Prinzip der sequentiellen Verarbeitung lässt sich gut mit Aktivitätsdiagrammen visualisieren.
- **Ablauf:** Filter nimmt Daten auf, transformiert sie und gibt sie an nächsten Filter weiter.
- **Beispiel:** Pipe-Operator in Unix: "ls -l | grep '2020'"

## 5 Folie 5: Software Design

### 5.1 Classdiagrams (Klassendiagramme)

#### Definition 5.1: Klasse

Beschreibung einer Menge an Objekten, die die selben Attribute, Operationen, Beziehungen und die selbe Semantik teilen

#### Definition 5.2: Assoziation

Strukturelle Beziehung eines Objekts mit einem anderen. Wenn eine Klasse eine Beziehung zu einer anderen hat, übernimmt es eine gewisse **Rolle**. Die **Multiplizität** gibt an, wie viele Objekte ein bestimmtes Objekt besitzt.

### 5.1.1 Attribute und Operationen

#### Attribute

- repräsentieren die strukturellen Merkmale oder Datenfelder einer Klasse.
- speichern Informationen, die den Zustand eines Objekts beschreiben.
- beim Domänenmodell sollten nur Datentypen verwendet werden, die direkt aus der fachlichen Domäne stammen.

#### Methoden (Operationen)

- beschreiben das Verhalten oder die Dienste, die ein Objekt einer Klasse ausführen kann.
- definieren die Schnittstelle, über die mit einem Objekt interagiert werden kann.
- werden in Domänenmodellen (Domain Models) nicht dargestellt, um den Fokus auf die fachliche Struktur statt auf technische Details zu legen.

#### Sichtbarkeit

- - private: nur in derselben Klasse
- + public: von jeder Klasse
- # protected: nur von derselben Klasse und deren Subklassen
- : von Klassen im selben Package

## 5.2 Aggregation und Komposition

#### Aggregation

- Beziehung, bei der ein Objekt ein anderes "besitzt" oder aus ihm besteht, aber beide Objekte eine eigene Lebensdauer haben.
- **Symbol:** Eine leere (weiße) Raute an der Seite des "Ganzen".
- **Eigenschaft:** Wenn das übergeordnete Objekt gelöscht wird, existieren die Teil-Objekte weiterhin.

#### Komposition

- **Symbol:** ausgefüllte (schwarze) Raute an der Seite des "Ganzen".
- **Eigenschaft:** wenn das "Ganze" zerstört wird, werden automatisch auch alle seine "Teile" vernichtet. Ein Teil kann zudem immer nur zu genau einem Ganzen gehören.

## 5.3 Inheritance

### 5.3.1 Generalisierung

- Beziehung zwischen einer allgemeinen Klasse (Superklasse) und einer spezialisierteren Klasse (Subklasse).
- **Konzept:** Die spezialisierte Klasse erbt alle Eigenschaften (Attribute) und Verhaltensweisen (Operationen) der allgemeinen Klasse.
- **Is-a Beziehung:** Man sagt, die Subklasse ist eine Instanz der Superklasse (z. B. eine Lecture ist eine CoursePart).
- **Darstellung:** Ein Pfeil mit einer leeren Dreiecksspitze, die von der spezialisierten Klasse zur allgemeinen Klasse zeigt.

### 5.3.2 Abstrakte Klasse

- dient als Vorlage für andere Klassen und kann selbst nicht direkt instanziert werden.
- **Zweck:** bündelt gemeinsame Merkmale für verschiedene Unterklassen, stellt aber allein kein vollständiges, reales Objekt dar.
- **Darstellung:** Name einer abstrakten Klasse wird in der UML üblicherweise kursiv geschrieben.

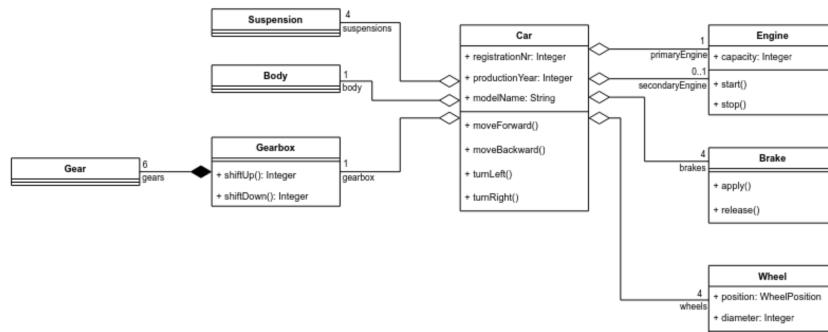


Abbildung 8: Class Diagram

## 5.4 Sequence Diagrams (Sequenzdiagramme)

- Zweidimensionales Diagramm, das **horizontal** die Rollen der einzelnen Komponenten und **vertikal** die Zeitachse darstellt.
- **Lebenslinien** (Lifelines): Rechtecke mit einer gestrichelte vertikale Linie nach unten, welche die Existenz des Objekts über die Zeit repräsentiert.
- **Nachrichten** (Messages): Horizontale Pfeile zwischen den Lebenslinien.
- **Durchgezogener Pfeil**: Ein Aufruf einer Operation (Synchron).
- **Gestrichelter Pfeil**: Die Antwort (Return) auf einen vorherigen Aufruf.
- **Aktivierungsbalken** (Execution Specification): Schmale Rechtecke auf der Lebenslinie, die anzeigen, wann ein Objekt gerade aktiv ist und eine Operation ausführt.

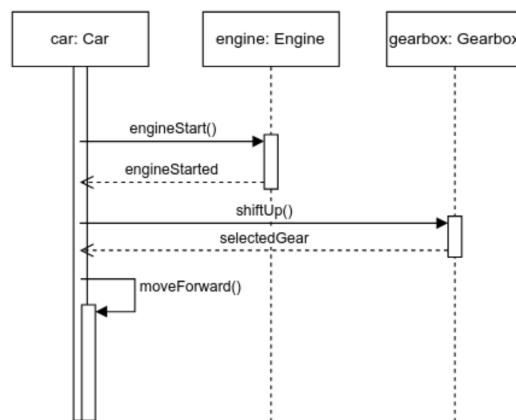


Abbildung 9: Sequence Diagram

# 6 Folie 6: Implementation

## Definition 6.1: Coding Conventions

Die grundlegende Motivation hinter Programmierrichtlinien ist, einen gut Lesbaren Code zu schreiben, bei dem man im Nachhinein noch Bock bekommt ihn zu lesen.

### 6.1 Code Formatierung

Um zu vermeiden, dass jeder Programmierer seinen eigenen Stil anwendet, gibt es Formatierungsregeln, die für jede Programmiersprache typisch sind:

- **Einrückung** (indentation): 4 Leerzeichen pro Ebene
- **Zeilenlänge**: auf 80 oder 100 Zeichen begrenzt
- **Zusätzliche Einrückung**: 8 Zeichen, aber auch nur wenn besonders lange Zeilen umgebrochen werden müssen
- **Zeilen Abstände**: Leere Zeilen zwischen Methoden und Attribute

### 6.2 Namenskonventionen (Rules of Naming)

Unerwünschte Namen	Erwünschte Namen
- einzelne Buchstaben als Namen - extrem lange Namen - Namen nur aus Sonderzeichen	- Klassen als Nomen, z.B. <code>Calculator</code> - Attribute <code>calculateButton</code> - Methoden als Verben, z.B. <code>getCalculator()</code> , <code>evaluate()</code> , <code>isZero()</code> , <code>hasChild()</code> , <code>setValue()</code>
- Synonyme für dieselbe Aktion (z.B. wenn man vermischt <code>delete</code> , <code>remvoe</code> , <code>clear</code> nutzen)	- Schreibweise für Attribute, Me- thoden Variablen und Parameter: <code>CamelCaseNotation</code>
- Abkürzungen (außer es handelt sich um äußerst nütliche)	- Schreibweise für Konstanten: <code>UPPER</code> <code>CASE NOTATION</code>
- Namen, die sich nur durch eine angehängte Zahl unterscheiden (z.B. <code>outputUser1</code> , <code>outputUser2</code> )	- Schreibweise für Packgae names: <code>lowercasenotation</code>

**Begriffspaare für Methoden:** add/remove, get/set, start/stop, lock/unlock, min/max, old/new, open/close

- **Gutes Beispiel:** Wenn es eine Methode `lightOn()` gibt, sollte das Gegenstück `lightOff()` heißen
- **Schlechtes Beispiel:** `List.add()` in Kombination mit `List.removeItem()`

### 6.3 Kommentare (Code Dokumentation)

Es sollte erst versucht werden den Code so zu schreiben, dass er weitestgehend selbsterklärend ist, nicht das der Kommentar noch überflüssig ist.

- Kommentare im **Quellcode** schreiben als in ein externes Dokument
- Kommentare **direkt während dem Programmieren** schreiben, da man da den Zusammenhang direkt versteht
- Kommentare dienen dazu Klassen und öffentliche Methoden direkt zu **spezifizieren**
- Kommentare dokumentieren unfertige stelle ('TODO')
- **Wichtige Regel:** Kommentare sollten den Code nicht umschreiben, sondern erklären, warum etwas getan wurde

### 6.4 Werkzeuge und Umgebung (Tools and Environments)

#### 6.4.1 Definitionen

- **Tool** (Werkzeug): ist eine Hilfsanwendung, um spezifische Aufgaben zu erleichtern, Prozesse zu automatisieren.
- **Umgebung** (Environments): Eine Sammlung von zusammengehörige Werkzeuge
- Der Überbegriff, für das Forschungsfeld, das sich mit der Automatisierung von Aktivitäten in der Softwareentwicklung beschäftigt, heißt **Computer-Aided Software Engineering**

#### 6.4.2 Übersicht gängiger Development Tools

- **Editoren:** Textbasiert (Word, Obsidian, Pages, usw...), Grafische Editoren (PowerPoint, Photoshop, Paint)
- **Übersetzer:** Assembler, Compiler

- **Versionskontrolle:** git, SVN (Dokumentation Änderungen am Code), CVS (Client-Server-Modell)
- **Tracking:** Aufgabenverwaltung mittels Github, Gitlab, Jira, usw.
- **Code-Bearbeitung:** Tools zur Code-Navigation
- **Qualitätssicherung:** Tools für die Definition, Generierung und Ausführung von Tests sowie für das Reporting
- **Analyse & Management:** Tools wie Debugger um code Statisch und Dynamisch zu analysieren
- **IDEs** (integrated development environments): IntelliJ, VS, Eclipse, usw.

## 7 Folie 7: Design Patterns

Design Patterns nach der "Gang of Four" (GoF) sind Lösungsschablonen für wiederkehrende Entwurfsprobleme in der objektorientierten Softwareentwicklung, die die Software robuster und wieder verwendbarer machen soll. Die Entwurf Muster werden in 3 Hauptkategorien unterteilt:

- **Strukturmuster** (Structural Patterns)
- **Erzeugungsmuster** (Creational Patterns)
- **Verhaltensmuster** (Behavioral Patterns)

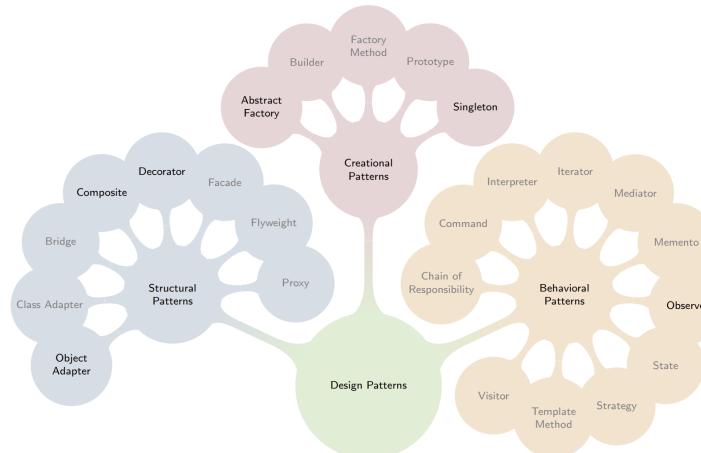


Abbildung 10: Design Patterns

## 7.1 Strukturmuster

Strukturmuster fassen Objekte und Klassen zu größeren Strukturen zusammen, um damit auch komplexe Zusammenhänge abbilden zu können. Da sich nicht alles mit Vererbung lösen lässt.

### 7.1.1 Object Adapter Pattern

#### Definition 7.1: Object Adapter Pattern

Wandelt die Schnittstelle einer Klasse in eine andere Schnittstelle um, die vom Client (also Nutzer der Klasse) erwartet wird

- **Wann/Warum anwenden:** Es ermöglicht die Zusammenarbeit und Wiederverwendung von Klassen, deren inkompatible Schnittstellen nicht direkt passend gemacht werden können
- **Beispiele**
  - Deutscher Föhn in den USA, Problem Steckdose ist Target (System erwartet US Stecker) aber deutscher Föhn ist Adaptee (nicht Kompatibel weil deutsch)
  - Was tun? Du baust den Föhn nicht um. Du steckst einen Adapter (Reiseadapter) dazwischen
  - Der Reiseadapter passt auf der einen Seite exakt in die US-Steckdose (request()) und leitet den Strom intern an den deutschen Stecker deines Föhns weiter ( specificRequest() ).

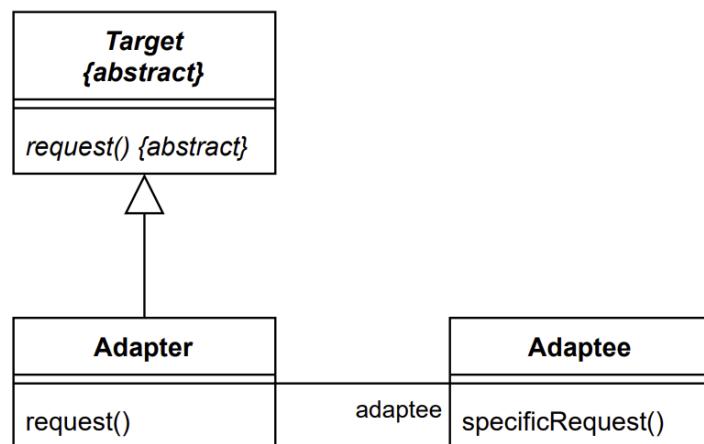


Abbildung 11: Object Adapter Pattern

### 7.1.2 Composite Pattern

#### Definition 7.2: Composite Pattern

Ordnet Objekte in einer Baumstruktur an und erlaubt es uns, einzelne Objekten und ganze Gruppen von Objekten im Code exakt gleich zu behandeln.

- **Problem:** Wenn wir mit einzelnen Objekten und Gruppen von Objekten arbeiten, wollen wir im Code nervige Fehlentscheidungen vermeiden ("Wenn es ein Einzelteil ist, mache X, wenn es eine Gruppe ist, mache Y")
- **Lösung:** Wir erstellen eine gemeinsame, abstrakte Oberklasse Components , die einen einheitlichen Zugriff auf alles ermöglicht

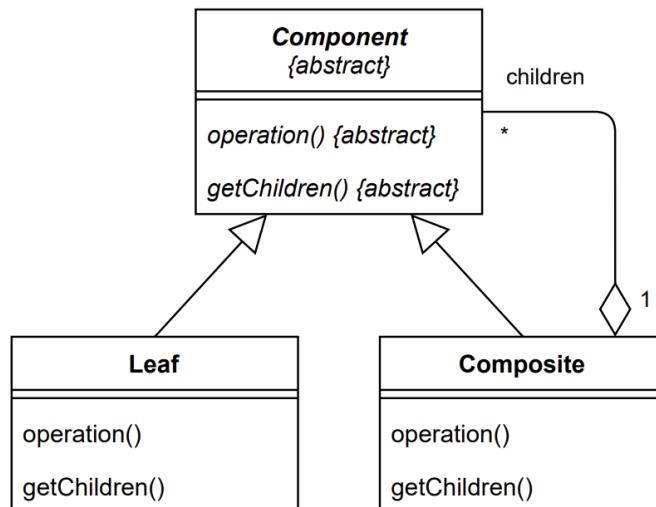


Abbildung 12: Composite Pattern

### 7.1.3 Decorator Pattern

#### Definition 7.3: Decorator Pattern

Ermöglicht es, einem Objekt zur Laufzeit neue Funktionen oder Verantwortlichkeiten hinzuzufügen, ohne den ursprünglichen zu verändern.

- **Problem:** Wenn wir Objekte flexibel mit neuen Fähigkeiten ausstatten, führt normale Vererbung schnell zur Explosion von Statischen Klassen.
- **Lösung:** Wir nutzen Decorators, die wie eine Hülle um das eigentliche Objekt gelegt werden. Deshalb wird dieses Muster auch oft "Wrapper" genannt.

- **Beispiele**

- Du Kaufst die einen Basis Kaffee ( ConcreteComponent )
- Jetzt möchtest du Milch dazu geben. Anstatt eine neue Tasse mit KaffeeMitMilch zu erzeugen, stülpt man einen Milch-Decorator ( ConcreteDecoratorA ) über deinen Basis Kaffee. Er fügt dann die Eigenschaften Milch und Aufpreis hinzu.
- Willst du noch Zucker, legst du einfach noch einen Zucker-Decorator ( ConcreteDecoratorB ) an
- Nach außen ist es für die Kasse ein einfaches Getränk, aber durch die ineinander verschachtelten Schichten hat es dynamisch neue Eigenschaften bekommen.

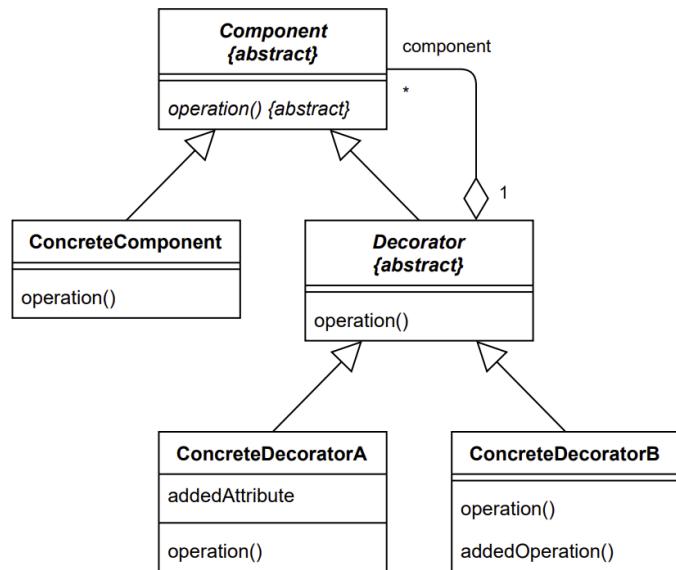


Abbildung 13: Decorator Pattern

## 7.2 Erzeugungsmuster

Bei Erzeugungsmustern dreht sich alles um das Erzeugen von Objekten.

### 7.2.1 Singleton Pattern

#### Definition 7.4: Singleton Pattern

Stellt sicher, dass von einer bestimmten Klasse immer nur exakt **ein einziges Objekt** existiert, und bietet einen globalen Zugriffspunkt auf genau dieses Objekt.

- **Vorteil:** Kontrolliert Zugriff auf einzelne Ressourcen.
- **Nachteil:** Schwerer zu Testen, weil globaler Zustand

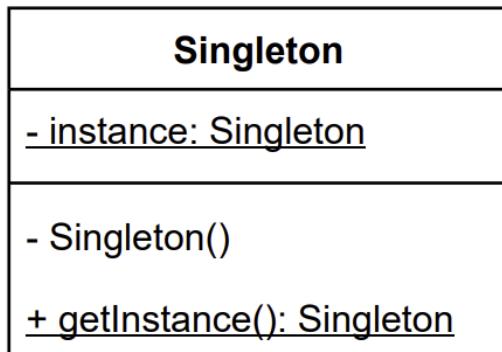


Abbildung 14: Singleton Pattern

### 7.2.2 Abstract Factory Pattern (aka: Kit)

#### Definition 7.5: Abstract Factory Pattern

Bietet eine Schnittstelle, um ganze Gruppen von verwandten oder voneinander abhängigen Objekten zu erstellen, ohne deren genaue Klasse im Code hart verdrahten zu müssen.

- **Vorteile:** Garantierte Konsistenz, heißt die Fabrik stellt sicher, dass alle erstellten Objekte einer Gruppe zwingend zusammenpassen
- **Nachteil:** Schwer Erweiterbar, erst muss der AbstractFactory verändert werden und in der Folge auch alle bereits bestehende konkreten Fabriken anpassen.

- Beispiel:

- Software für Möbelhaus: Du brauchst Stühle und Tische, das aber in zwei Stilen: "Modern" und "Viktorianisch".
- Anstatt bei jedem einzelnen Möbelstück zu prüfen, ob es stilistisch zu dem Rest des Zimmers passt, nutzt man Fabriken.
- Du hast eine ModerneMoebelFabrik ( ConcreteFactory1 ) und ViktorianischeMoebelFabrik ( ConcreteFactory2 )
- Wenn du jetzt dem System sagst, "Benutze die Moderne Fabrik" und dann "Erstelle Stuhl" und "Erstelle Tisch" aufrufst, kannst du dir 100% sicher sein, dass beide Möbelstücke im modernen Stil erhältst.

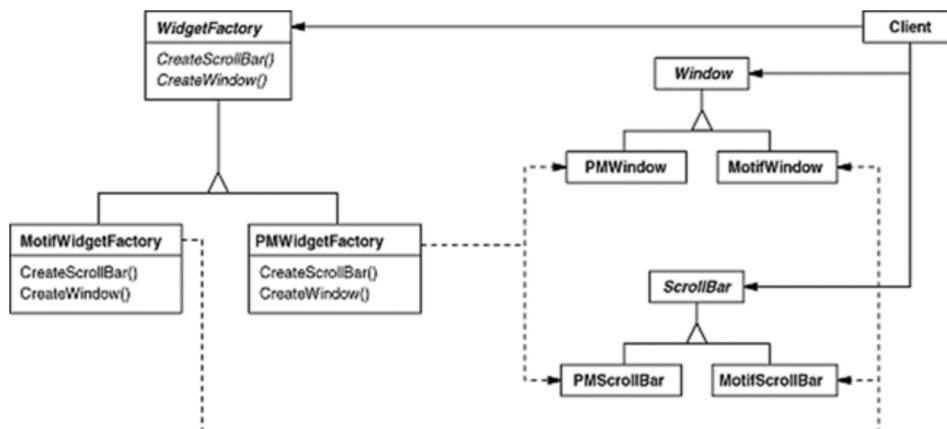


Abbildung 15: Abstract Factory Pattern

### 7.3 Verhaltensmuster

Beschreiben die Kommunikation und Interaktion zwischen Objekten.

### 7.3.1 Observer Pattern

#### Definition 7.6: Observer Pattern

Definiert eine **one-to-many-dependency** zwischen Objekten, sodass bei der Zustandsänderung eines Kernobjekts ( Subject/Observable ) alle anhängigen Objekte ( Observers ) automatisch benachrichtigt werden.

- Das Muster wird auch "Publish-Subscribe" genannt. Es gibt ein Objekt, das die Daten hält und eine Liste von Beobachtern die über die Änderung informiert werden.
- **Vorteile:** Lose Kopplung bedeutet, dass Observable kennt seine Beobachter im Detail, man kann jederzeit neue Beobachter hinzufügen.
- **Nachteile:** Wenn ein Beobachter gelöscht wird, aber vergessen wird ihn mit removeObserver() zu entfernen, wird er weiterhin im Speicher festgehalten. Dadurch wird das Programm auf dauer langsamer
- **Beispiele:**
  - Stell dir einen YouTube-Kanal vor. Dieser Kanal ist dein Observable
  - Tausende andere Nutzer sind die Observer
  - Du klickst auf Abonnieren (das System ruft addObserve() auf)
  - Wenn der YTber nun ein neues Video hochlädt, muss er nicht jeden neuen Abonnenten einzeln aufrufen. Das System ruft einfach intern notifyObservers() auf und ihr alle bekommt gleichzeitig eine Push Benachrichtigung update() .

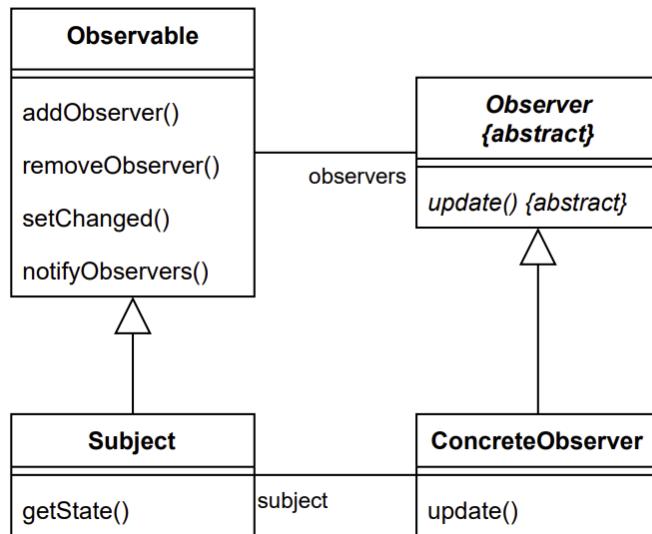


Abbildung 16: Observer Pattern

## 8 Folie 8: Quality Assurance

### Definition 8.1: Qualität

Qualität ist die Gesamtheit der Eigenschaften und Merkmale eines Produkts oder Prozesses, die auf dessen Eignung hinsichtlich der gegebenen Anforderungen hinweisen.

### Definition 8.2: Qualitätssicherung

Alle Aktivitäten mit dem Ziel, die Qualität zu verbessern

### Qualitätssicherung

- **Konstruktiv:** Entstehen von Fehlern im Vorhinein vermeiden
- **Analytisch:** Untersuchen des fertigen Produkts auf Fehler
- **Organisatorisch:** beziehen sich weniger auf den Code selbst, sondern auf die Rahmenbedingungen und Prozesse der Softwareentwicklung

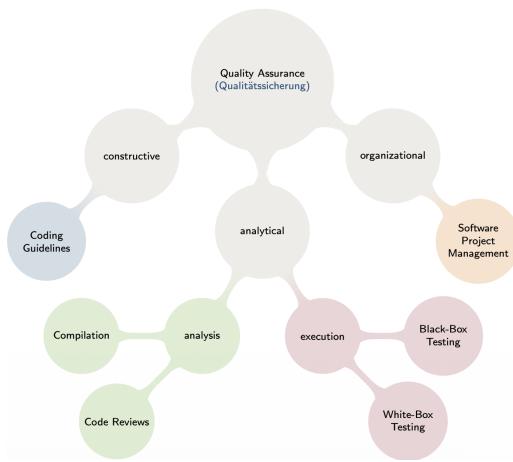


Abbildung 17: Quality Assurance

## 8.1 Expectation on Quality

### Niedrige Erwartungen

- Fehlerbehaftete Vorgänger: Wenn die vorherige Software bereits verbuggt und unzuverlässig war.
- Keine Überraschung bei Ausfällen: Nutzer sind nicht überrascht, wenn die Software versagt.
- Nutzen überwiegt Aufwand: Die Vorteile eines neuen Systems überwiegen die Kosten für die Fehlerbehebung bzw. Wiederherstellung.
- Günstiger Preis: Die Software-App ist sehr preiswert.
- Ökonomie der Qualität: Kunden sind oft bereit, Software trotz Problemen zu akzeptieren, weil die Kosten für die Nichtnutzung der Software höher sind als die Kosten für das Umgehen der Probleme (Workarounds).

### Hohe Erwartungen

- Etablierte Produkte: Wenn ein Softwareprodukt am Markt etablierter und bekannter wird.
- Fokus auf Zuverlässigkeit: Die Nutzer erwarten in diesem Stadium, dass die Software wesentlich zuverlässiger funktioniert.

## 8.2 Product Quality

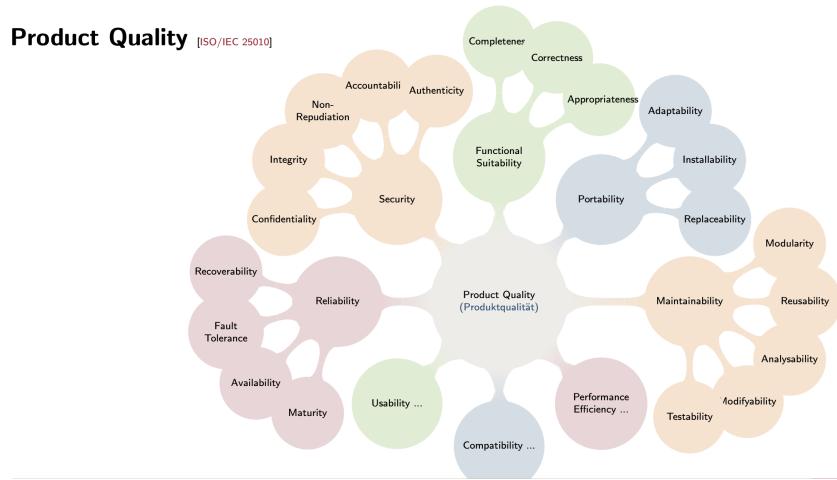


Abbildung 18: Product Quality

## 8.3 Quality in Use

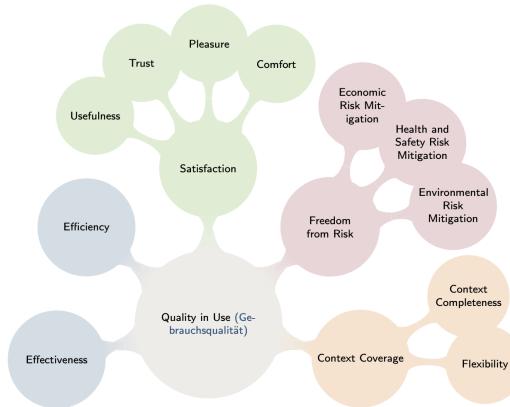


Abbildung 19: Quality in Use

## 8.4 Software Testing

### Definition 8.3: Software Testing

Software-Tests dienen dazu, nachzuweisen, dass das Programm seine beabsichtigte Funktion erfüllt und um Programmierfehler aufzudecken, bevor es in Betrieb genommen wird.

- **Validation Testing:** Validierungstests demonstrieren sowohl dem Entwickler als auch dem Kunden, dass die Software ihre Anforderungen erfüllt.
- **Defect Testing:** Defekttests dienen dazu, Eingabesequenzen zu finden, bei denen das Verhalten der Software fehlerhaft, unerwünscht oder nicht spezifikationskonform ist.

### V&V

Merkmal	Verifizierung (Verification)	Validierung (Validation)
<b>Leitfrage</b>	„Bauen wir das Produkt richtig?“	„Bauen wir das richtige Produkt?“
<b>Fokus</b>	Prüfung gegen die Spezifikationen und das Design.	Prüfung gegen die tatsächlichen Kundenanforderungen.
<b>Ziel</b>	Nachweis der technischen Korrektheit und Einhaltung von Standards.	Nachweis, dass die Software den beabsichtigten Nutzen für den Anwender erbringt.
<b>Zeitpunkt</b>	Findet während des gesamten Entwicklungsprozesses statt.	Findet primär nach Abschluss von (Teil-)Entwicklungen statt.

Tabelle 2: Vergleich von Verifizierung und Validierung im Software-Testprozess

## 8.5 Stages of Testing

1. **Development Testing** (Entwicklungstests): Tests während der Entwicklung, um Fehler frühzeitig zu entdecken.
2. **Release-Testing:** Ein separates Team testet eine vollständige Version des Systems vor der Veröffentlichung.

3. **User Testing** (Anwendertest): Nutzer oder potenzielle Kunden testen das System in ihrer eigenen Umgebung.
  - **Manuelles Testen:** Ein Tester führt das Programm mit Testdaten aus und vergleicht das Ergebnis mit den Erwartungen.
  - **Automatisiertes Testen:** Tests werden in einem Programm kodiert, das bei jeder Systemänderung automatisch ausgeführt werden kann.

## 8.6 Code-Reviews

- **Grundidee:** Verbesserung der Qualität durch Feedback von anderen Programmierern.
- Wird häufig mit Hilfe von Qualitäts-Checklisten durchgeführt.
- **Qualitätskriterien:** Funktionalität, Verständlichkeit, Wartbarkeit, Einhaltung von Programmierrichtlinien, Design-Patterns.
- **Auswahl der Reviewer:** Erfolgt nach Kriterien wie Vertrautheit mit dem Code, Verfügbarkeit und Fachwissen.

### Wichtige Aspekte:

- Man kann seinen eigenen Code nicht objektiv selbst reviewen.
- Reviewer benötigen Programmiererfahrung und Kenntnis des Codes.
- Feedback sollte zeitnah und konstruktiv gegeben werden.
- Es sollten nur Änderungen begutachtet werden, keine zu großen Codemengen auf einmal.

*Code-Reviews auf GitHub:* Ein **Pull Request** eröffnet die Diskussion über die vorgeschlagenen Änderungen.

## 8.7 White-Box-Testing (Strukturtests)

### 8.7.1 Testfälle

- **Systematic Test**

1. Testaufbau ist genau definiert
2. Eingaben werden systematisch gewählt

3. Ergebnisse werden dokumentiert und nach vorab definierten Kriterien bewertet

- **Testfall:** Ein Test besteht aus mehreren Testfällen. Ein Testfall umfasst Eingabewerte für eine einzelne Ausführung und die dazugehörigen erwarteten Ausgabewerte. Ein **vollständiger Test** (Exhaustive Test) würde alle theoretisch möglichen Eingaben prüfen.

### Vollständiges Testen in der Praxis

```
boolean a b c;  
int i j;  
bla(a, b, c) has  $2^3 = 8$  possible inputs;  
blub(i, j) has  $2^{32} \cdot 2^2 = 2^{64}$  possible inputs; (585 years)
```

#### 8.7.2 Test-case Design (Testfallentwurf)

**Ziel:** Entdeckung einer großen Anzahl an Fehlern mit möglichst wenigen Testfällen.

- Ein Testlauf ist **positiv**, wenn er einen Fehler aufdeckt.
- Er ist **erfolgreich**, wenn er einen bisher unbekannten Fehler findet.

#### Idealer Testfall:

- **repräsentativ:** steht für eine große Anzahl möglicher Testfälle.
- **fehlersensitiv:** hat eine hohe Wahrscheinlichkeit, einen Fehler zu finden.
- **nicht redundant:** prüft nichts, was andere Testfälle bereits abdecken.

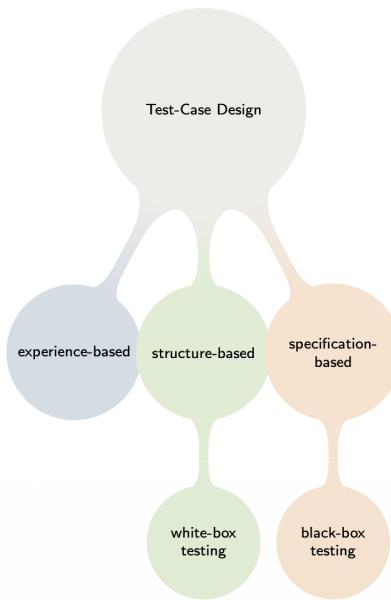


Abbildung 20: Test Case Design

Language	$13 \bmod 3$	$-13 \bmod 3$	$13 \bmod -3$	$-13 \bmod -3$
C	1	-1	1	-1
Go	1	-1	1	-1
PHP	1	-1	1	-1
Rust	1	-1	1	-1
Scala	1	-1	1	-1
Java	1	-1	1	-1
Javascript	1	-1	1	-1
Ruby	1	2	-2	-1
Python	1	2	-2	-1

Abbildung 21: Modulo in different Programming Languages

### 8.7.3 White-Box-Testing (Strukturtest)

- Nutzt die interne Struktur des Testobjekts.
- Ziel ist die Überdeckung struktureller Elemente.
- Der Code wird in einen **Kontrollflusssgraphen** übersetzt.

- Konkrete Testfälle (Eingaben) werden aus logischen Testfällen (Bedingungen) abgeleitet, die wiederum aus Pfaden im Graphen stammen.

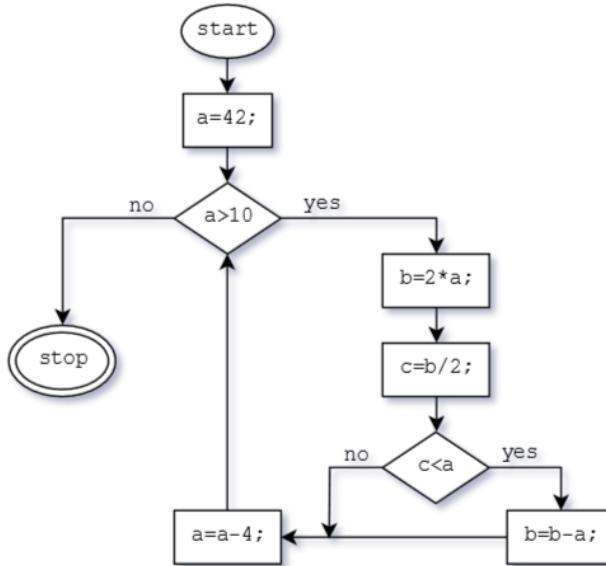


Abbildung 22: Kontrollfluss Graph

#### 8.7.4 Coverage Criteria (Überdeckungskriterien)

1. **Statement Coverage** (Anweisungsüberdeckung): Jede Anweisung im Code wird mindestens einmal ausgeführt.
2. **Branching Coverage** (Zweigüberdeckung): Beinhaltet Anweisungsüberdeckung; zudem werden bei jeder Verzweigung alle möglichen Ausgänge durchlaufen.
3. **Term Coverage** (Termüberdeckung): Beinhaltet Zweigüberdeckung; zusätzlich wird jede einzelne atomare Bedingung (jeder Term) innerhalb einer zusammengesetzten Bedingung mindestens einmal mit dem Wert true und einmal mit dem Wert false getestet

**In der Praxis:** 100% Anweisungsüberdeckung ist oft nicht möglich (z.B. durch toten Code oder unerreichbare Fehlerbehandlung).

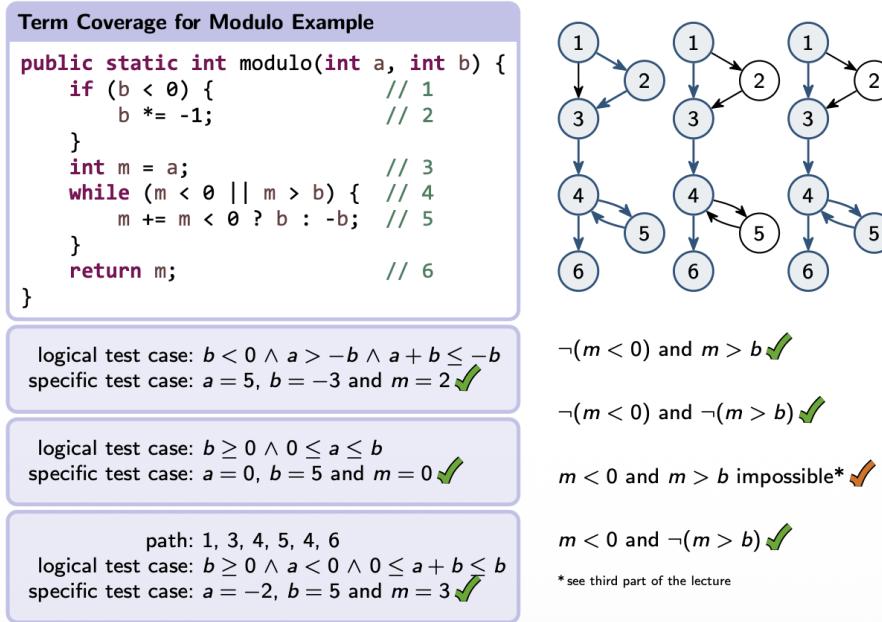


Abbildung 23: White Box Test Coverage

## 8.8 Black-Box-Tests Testing

Motivation:

- Quellcode ist nicht immer verfügbar.
- Spezifikationen wurden bisher oft nicht in den Test einbezogen.
- Ungültige Eingaben werden oft vergessen.
- Fehler sind nicht gleichmäßig verteilt.

### Black-Box-Testing (Funktionstest)

- Testfall Design basierend auf Spezifikation
- Source-Code und dessen innere Struktur wird ignoriert

#### 8.8.1 Äquivalenzklassentests

- **Idee:** Eingaben und Ausgaben in Klassen unterteilen, die sich bezüglich des Testens gleich verhalten.

- **Annahme:** Äquivalente Testfälle finden dieselben Fehler; daher genügt ein Repräsentant pro Klasse.

Equivalence Class Testing			
JavaDoc Specification for Modulo Example			
/** * Computes the remainder of the * Euclidean division of a by b. In * contrast to the Java version a % b, * the output will always be positive. * Throws ArithmeticException when b is * equal to 0. * * @param a dividend * @param b divisor != 0 * @return remainder r with 0 <= r < b */	a < 0	X	
public static int modulo(int a, int b) {	a ≥ 0		X
	b < 0	X	
	b > 0		X
	b = 0		X
	m = 0	X	
	m > 0		X
	exception		X
	input a	-3	1
	input b	-3	2
	expected output	0	1
	result	0 ✓	1 ✓
			timeout ✘

Abbildung 24: Equivalence Class Testing

### 8.8.2 Grenzwerttests (Boundary Testing)

- Erweiterung des Äquivalenzklassentests.
- **Ziel:** Erfahrungsgemäß treten Fehler oft an den Rändern von Wertebereichen auf.
- Für jede Äquivalenzklasse werden der kleinste, ein typischer und der größte Wert geprüft.

### 8.8.3 Gründe für positive Testfälle (Fehlerfunde)

Wenn ein Test einen Fehler anzeigt, kann dies verschiedene Ursachen haben:

- Ein tatsächlicher Fehler im Code.
- Ein fehlerhafter Testfall (Eingabe und Erwartung passen nicht zusammen).
- Wechselwirkungen mit anderen Programmen oder Bibliotheken.

- Fehler im Compiler oder im Betriebssystem/Treiber.
- Hardwaredefekte.
- Speichermangel oder Endlosschleifen.
- Bitflips (z.B. durch kosmische Strahlung).

## 9 Folie 9: Development Process

Ein Software Development Process ist wie das **Rezept** oder **Bauplan** für ein Projekt. Es ist eine Menge von zusammenhängenden **Aktivitäten**, die zur Erstellung eines Softwaresystems führen. Er definiert **Produkte/Liefergegenstände** (Deliverables), **Rollen** und **Verantwortlichkeiten** sowie **Vor- und Nachbedingungen** (conditions) für bestimmte Phasen.

### 9.1 Das Wasserfallmodell (The Waterfall Model)

Die Phasen überlappen sich nicht, sondern werden strikt sequenziell durchlaufen.

#### 9.1.1 Die 5 Phasen

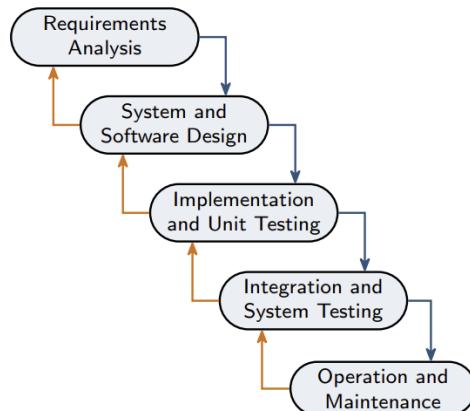


Abbildung 25: Waterfall Model

- **Erklärung:** Die Arbeit fließt wie bei einem Wasserfall von oben nach unten durch folgende Stationen:  
**Anforderungsanalyse → Design → Implementierung & Modultest → Integration & Systemtest → Betrieb & Wartung.** Die

Pfeile nach oben im Diagramm zeigen, dass man bei Fehlern zwar eine Stufe zurückspringen kann, dies aber eigentlich nicht der normale Fluss ist.

- **Vorteile:** Es ist sehr leicht zu verstehen, zu steuern und zu managen. Es eignet sich besonders gut, wenn Hard- und Software parallel entwickelt werden, da man ähnliche Phasenmodelle nutzen kann.
- **Nachteile:** Frühe Phasen sind isoliert. Änderungen an vorherigen Schritten sind im Nachhinein extrem schwer umsetzbar. Wenn Anforderungen zu früh eingefroren werden, erhält der Kunde am Ende oft Software, die er so gar nicht will.

## 9.2 Das V-Modell (V-Model for Extensive Testing)

Das V-Modell ist eine Erweiterung des Wasserfallmodels. Jeder Planungsphase auf der linken Seite wird direkt eine passende Testphase auf der rechten Seite gegenübergestellt.

### 9.2.1 Die 4 Teststufen

1. **Unit Testing:** Testet einzelne isolierte Einheiten (z. B. Klassen oder Methoden). Wird meist vom Entwickler selbst durchgeführt und automatisiert.
2. **Integration Testing:** Getestete Komponenten werden (z.B. zu Subsystemen) zusammengefügt. Findet Schnittstellenprobleme und Kommunikationsfehler zwischen den Komponenten.
3. **System Testing:** Das komplette, integrierte System wird gegen die Systemanforderungen (Pflichtenheft) getestet, um unvorhergesehene Interaktionen aufzudecken.
4. **Acceptance Testing:** Der finale Test durch den Kunden vor der Inbetriebnahme. Das System wird mit realen Daten gegen die Nutzeranforderungen (Lastenheft) validiert.

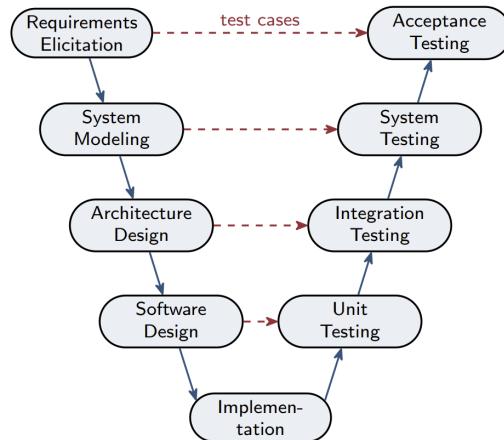


Abbildung 26: V-Modell

- **Erklärung:** Auf der linken Seite wandert man im Diagramm abwärts von der groben Planung bis zum fertigen Code. Auf der rechten Seite wandert man wieder aufwärts und durchläuft dabei vier genau abgestimmte Teststufen.
- **Gestrichelte Linien:** Schon während man links z.B. die Systemanforderungen definiert, schreibt man gleichzeitig die Testfälle für den späteren Systemtest auf der rechten Seite.
- **Vorteile:** Da die Tests schon während der Planung entwickelt werden, entdeckt man Planungsfehler viel früher. Es gibt am Ende weniger böse Überraschungen.
- **Nachteile:** Wie das Wasserfallmodell ist es recht schwerfällig. Wenn der Kunde auf halbem Weg seine Meinung ändert, ist das sehr aufwendig, weil alle Dokumente und Tests auf der linken und rechten Seite angepasst werden müssen.

### 9.3 Motivation

Agile Methoden entstanden, weil:

- Anforderungen oft nicht stabil sind
- Märkte sich schnell ändern
- Software schnell geliefert werden muss
- klassische Modelle zu spät Ergebnisse liefern

### 9.3.1 Agile (Development) Methods

1. Spezifikation, Design und Implementierung sind miteinander verflochten
2. Jedes Inkrement wird von den Stakeholdern (z. B. Endnutzern) spezifiziert und bewertet
3. Es wird umfangreiche Tool-Unterstützung genutzt

### 9.3.2 Agile Manifesto

1. **Kundenzufriedenheit durch frühe und kontinuierliche Lieferung:** Wichtig ist nicht Perfektion am Ende, sondern früh nutzbare Software. Der Kunde bekommt regelmäßig Mehrwert.
2. **Änderungen willkommen heißen – auch spät:** Anforderungen ändern sich. Agile Prozesse nutzen Änderungen als Vorteil statt sie zu bekämpfen.
3. **Häufig funktionierende Software liefern:** Kurze Iterationen (Wochen statt Monate). Fortschritt wird durch reale Software sichtbar, nicht durch Dokumente.
4. **Business und Entwickler arbeiten täglich zusammen:** Fachseite und Technik dürfen nicht getrennt sein. Kontinuierliche Abstimmung reduziert Missverständnisse.
5. **Motivierte Individuen und Vertrauen:** Teams sollen eigenverantwortlich arbeiten. Man schafft gute Rahmenbedingungen statt Mikromanagement.
6. **Direkte Kommunikation ist am effektivsten:** Face-to-Face-Kommunikation vermeidet Informationsverlust. Schnelle Abstimmung ist effizienter als lange Dokumentketten.
7. **Funktionierende Software ist das Maß für Fortschritt:** Nicht Planung, nicht Dokumentation, sondern lauffähige Software zählt.
8. **Nachhaltiges Entwicklungstempo:** Kein dauerhaftes „Überarbeiten“. Team, Sponsor und Nutzer sollen langfristig konstant arbeiten können.

9. **Technische Exzellenz und gutes Design:** Sauberer Code und gute Architektur machen Anpassungen einfacher. Agil heißt nicht „schnell und chaotisch“.
10. **Einfachheit:** So wenig wie nötig bauen. Unnötige Komplexität vermeiden.
11. **Selbstorganisierende Teams:** Teams entscheiden selbst, wie sie arbeiten. Gute Architektur entsteht aus Kompetenz und Zusammenarbeit.
12. **Regelmäßige Reflexion:** Das Team überprüft regelmäßig: Was lief gut? Was nicht? Dann wird der Prozess angepasst.

#### **Wichtigste Aussagen:**

- Frühe und kontinuierliche Lieferung von lauffähiger Software
- Änderungen sind willkommen, auch spät im Projekt
- Kurze Iterationen
- Enge Zusammenarbeit zwischen Business und Entwicklung
- Funktionierende Software ist Hauptmaß für Fortschritt
- Teams reflektieren regelmäßig und verbessern sich

#### **9.3.3 Scrum**

##### **Zentrale Scrum Elemente**

- **Product Backlog:** Gesamtliste aller Anforderungen (User Stories), priorisiert vom Product Owner.
- **Sprint:** Entwicklungsiteration (meist 2–4 Wochen).
- **Sprint Backlog:** Teilmenge der User Stories für den aktuellen Sprint.
- **Produktinkrement:** Am Ende jedes Sprints entsteht ein potenziell auslieferbares Produktstück.
- **Daily Scrum:** Kurzes tägliches Team-Meeting zur Abstimmung.
- **Velocity:** Maß dafür, wie viel Arbeit ein Team pro Sprint schafft.

## Scrum Rollen

- **Development Team:** Selbstorganisierend, max. ca. 7 Personen.
- **Product Owner:** Verantwortlich für Priorisierung und Geschäftswert.
- **Scrum Master:** Stellt sicher, dass Scrum korrekt angewendet wird und schützt das Team.

## Scrum Terme

1. **Product Backlog:** Die vollständige Liste aller Anforderungen, Features und Aufgaben für das Produkt. Wird vom Product Owner priorisiert. Enthält meist User Stories. → Alles, was irgendwann umgesetzt werden soll.
2. **Sprint:** Eine feste Entwicklungsiteration, meist 2–4 Wochen. Am Ende steht ein fertiges, nutzbares Ergebnis. → Ein klar abgegrenzter Arbeitszyklus.
3. **Sprint Backlog:** Die Auswahl an Product-Backlog-Einträgen, die im aktuellen Sprint umgesetzt werden. Wird vom Development Team geplant. → Das konkrete Arbeitsziel für die nächsten Wochen.
4. **(Potentially Shippable) Product Increment:** Das Ergebnis eines Sprints. Muss vollständig implementiert, getestet und integrierbar sein. → Theoretisch sofort auslieferbar.
5. **Daily Scrum:** Tägliches, kurzes Meeting (Stand-up). Ziel: Fortschritt synchronisieren und Hindernisse erkennen.
  - Was habe ich gestern gemacht?
  - Was mache ich heute?
  - Gibt es Blocker?
6. **Velocity:** Maß dafür, wie viel Arbeit ein Team pro Sprint schafft. Hilft bei der Planung zukünftiger Sprints. → Durchschnittliche Leistungsfähigkeit des Teams.

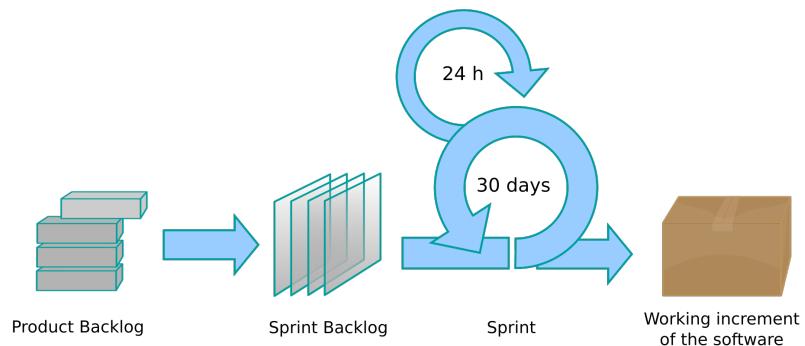


Abbildung 27: Scrum Process

#### 9.3.4 Vor- und Nachteile

##### Vorteile

- Flexibel bei sich ändernden Anforderungen
- Transparenz durch kurze Iterationen
- Frühes Feedback
- Gute Teamkommunikation

##### Nachteile

- Schwer skalierbar auf große Teams
- Starke Kundenbeteiligung nötig
- Dokumentation nicht automatisch im Fokus
- Problematisch bei festen Vertragsmodellen
- Für stark regulierte Systeme oft ungeeignet

## 10 Folie 10: Project Management

### 10.1 Definition und Ziele

Ein Software-Entwicklungsprojekt ist eine zeitlich begrenzte Aktivität mit festem Start- und Enddatum. Es verfolgt Ziele wie die Erstellung oder Modifikation von Softwareprodukten, die Vorbereitung zukünftiger Projekte oder

den Gewinn von Wissen. Ein Projekt gilt als erfolgreich, wenn diese Ziele weitgehend erfüllt werden.

Das Projektmanagement hat dabei folgende Kernaufgaben:

- Lieferung der Software zum vereinbarten Zeitpunkt.
- Einhaltung des Budgets.
- Bereitstellung von Software, die den Kundenerwartungen entspricht.
- Aufrechterhaltung eines koordinierten Entwicklungsteams.

Die Durchführung hängt maßgeblich von Faktoren wie der Unternehmensgröße, dem Kunden, der Softwareart und dem gewählten Entwicklungsprozess ab.

## 10.2 Risikomanagement

Das Risikomanagement ist eine der Hauptaufgaben der Projektleitung. Es befasst sich systematisch mit potenziellen Problemen in den Bereichen Zeitplan, Budget und Qualität.

### 10.2.1 Risikoklassifikation

- **Projektrisiken:** Beeinflussen den Zeitplan oder die Ressourcen (z. B. Personalfluktuation, Hardwareverfügbarkeit).
- **Produktrisiken:** Beeinflussen die Qualität oder Performance der Software (z. B. Anforderungsänderungen, dringendes Refactoring).
- **Geschäftsrisiken:** Beeinflussen die Organisation oder den Markterfolg (z. B. Produktkonkurrenz).

### 10.2.2 Risikostrategien und Phasen

Zur Bewältigung werden Strategien wie **Vermeidung** (Eintrittswahrscheinlichkeit senken), **Minimierung** (Auswirkungen reduzieren) und **Notfallpläne** eingesetzt. Der Prozess unterteilt sich in Identifikation, Analyse, Planung und kontinuierliche Überwachung.

Risiko	Risiko-Typ	Beschreibung
Personalfliktuation	Projekt	Erfahrene Mitarbeiter verlassen das Projekt
Managementwechsel	Projekt	Wechsel in der Organisationsleitung mit anderen Prioritäten
Hardware nicht verfügbar	Projekt	Notwendige Hardware wird nicht rechtzeitig geliefert
Anforderungsänderungen	Projekt & Produkt	Mehr Änderungen als erwartet
Größenunterschätzung	Projekt & Produkt	Systemgröße wurde unterschätzt
CASE tools mit schlechter Leistung	Projekt	Unterstützende CASE-Tools funktionieren nicht wie erwartet
Produktkonkurrenz	Geschäft	Konkurrenzprodukt erscheint vor Fertigstellung
Dringendes Refactoring	Produkt	Code-/Designqualität verschlechtert sich und muss verbessert werden

Tabelle 3: Mögliche Software-Risiken

Risk	Probability	Severity
Organisational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff are ill at critical times in the project.	Moderate	Catastrophic
Software components which should be reused contain defect which limit their functionality.	Moderate	Serious

Abbildung 28: Risiko Analyse

### 10.3 Personalmanagement

Mitarbeiter sind das wichtigste Kapital. Neben technischen Fähigkeiten sind Soft Skills zur Motivation und Teamführung entscheidend. Wichtige Faktoren für ein gutes Management sind:

- **Konsistenz:** Faire Behandlung aller Beteiligten.
- **Respekt:** Wertschätzung unterschiedlicher Fähigkeiten.

- **Einbeziehung:** Berücksichtigung aller Meinungen.
- **Ehrlichkeit:** Transparenz über eigene Fähigkeiten und die Teamleistung.

## 10.4 Projekt- und Terminplanung

Die Planung erfolgt in Phasen, beginnend mit der Angebotsphase, in der Ressourcen geprüft und Preise kalkuliert werden. Während des Projekts wird der Plan basierend auf neuen Erkenntnissen stetig verfeinert.

### 10.4.1 Visualisierungstools

- **Gantt Chart:** Zeitachse auf der x-Achse, Aktivitäten auf der y-Achse. Zeigt Fortschrittsbalken und Abhängigkeiten.
- **Netzplan (Network Diagram):** Ein gerichteter azyklischer Graph (DAG), bei dem Knoten Aufgaben und Kanten Abhängigkeiten darstellen.
- **Metra-Potenzial-Methode (MPM):** Ermöglicht die Berechnung von frühestem/spätestem Start- und Endzeitpunkt sowie Pufferzeiten.

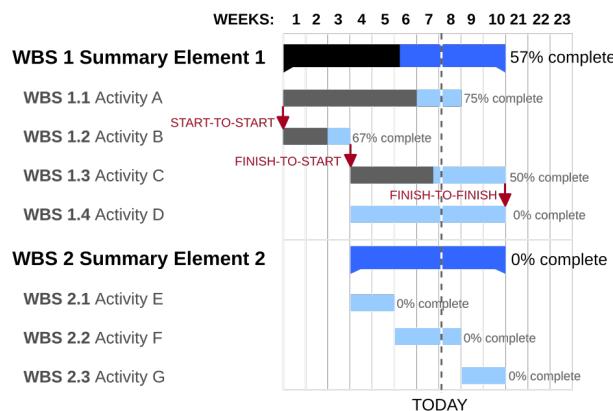


Abbildung 29: Gantt Chart

### 10.4.2 Konstruktion eines Netzplans (Network Diagram)

- **Schritt 1: Definition der Knotenstruktur**  
Jeder Vorgang im Netzplan wird durch einen Knoten repräsentiert, der folgende Standardwerte enthält:

- $DA$ : Dauer der Aktivität
- $FAZ$ : Frühestmöglicher Anfangszeitpunkt
- $FEZ$ : Frühestmöglicher Endzeitpunkt
- $SAZ$ : Spätestmöglicher Anfangszeitpunkt
- $SEZ$ : Spätestmöglicher Endzeitpunkt
- $GP$ : Gesamtpuffer ( $GP = SAZ - FAZ$  oder  $GP = SEZ - FEZ$ )

- **Schritt 2: Forward-Pass (Vorwärtsrechnung)**

Ziel ist die Ermittlung des frühestmöglichen Projektendes. Man rechnet vom Start zum Ziel:

- Setze den  $FAZ$  des Startknotens auf 0.
- Berechne  $FEZ = FAZ + DA$ .
- Der  $FAZ$  eines Nachfolgers entspricht dem  $FEZ$  des Vorgängers.
- **Maximum-Regel:** Bei mehreren Vorgängern gilt:  $FAZ_{Nachfolger} = \max(FEZ_{Vorgänger1}, FEZ_{Vorgänger2}, \dots)$ .

- **Schritt 3: Backward-Pass (Rückwärtsrechnung)**

Ziel ist die Ermittlung der spätestmöglichen Zeitpunkte. Man rechnet vom Ziel zurück zum Start:

- Setze den  $SEZ$  des Endknotens gleich seinem berechneten  $FEZ$ .
- Berechne  $SAZ = SEZ - DA$ .
- Der  $SEZ$  eines Vorgängers entspricht dem  $SAZ$  des Nachfolgers.
- **Minimum-Regel:** Bei mehreren Nachfolgern gilt:  $SEZ_{Vorgänger} = \min(SAZ_{Nachfolger1}, SAZ_{Nachfolger2}, \dots)$ .

- **Schritt 4: Analyse der Pufferzeiten und des Kritischen Pfads**

- Berechne den Gesamtpuffer  $GP$  für jeden Knoten.
- Alle Knoten mit  $GP = 0$  bilden den **Kritischen Pfad**.
- Der  $FEZ$  des letzten Knotens gibt an, wann das Projekt frühestens fertiggestellt werden kann.

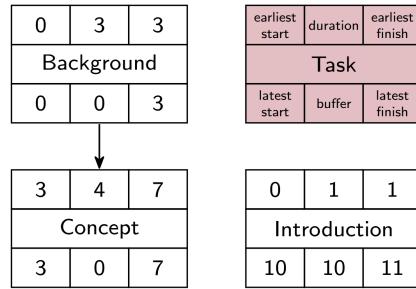


Abbildung 30: Network Diagram

## 11 Folie 11: Configuration Management

### 11.1 Konfigurationsmanagement und Versionskontrolle

„Konfigurationsmanagement befasst sich mit den Richtlinien, Prozessen und Werkzeugen zur Verwaltung sich ändernder Softwaresysteme.“

#### 11.1.1 Vier Aktivitäten im Konfigurationsmanagement

1. **Versionskontrolle / Versionsverwaltung:** Verwalten mehrerer Versionen, ermöglichen gleichzeitiger Änderungen.
2. **Systembau:** Sammeln, Kompilieren und Verknüpfen von Komponenten zu ausführbaren Systemen.
3. **Änderungsmanagement:** Verfolgen von Änderungsanträgen und Planung, ob und wann sie umgesetzt werden.
4. **Release-Management:** Vorbereitung neuer Releases und Verwaltung bestehender Releases.

### 11.2 Entwicklungsphasen

- **Entwicklungsphase:** Hinzufügen neuer Funktionalität.
- **Systemtestphase:** Interne Releases, Bug fixes, Leistungsverbesserungen, Security fixes. Keine neue Funktionalität.
- **Release-Phase:** Bearbeitung von Bug reports und Funktionsanfragen von Nutzerinnen, Nutzern oder Kundinnen und Kunden.

Oft existieren mehrere Versionen gleichzeitig zu unterschiedlichen Zeitpunkten.

## 11.3 Versionskontrolle

### 11.3.1 Ziele der Versionskontrolle

- **Zusammenarbeit:** Synchronisierung von Dateien und Ordnern mit anderen Nutzerinnen und Nutzern.
- **Vergleich:** Vergleich der eigenen Version mit anderen Versionen.
- **Zusammenführen:** Zusammenführen von Dateien, die von mehreren Personen bearbeitet wurden.
- **History:** Zugriff auf ältere Versionen und Nachverfolgung von Änderungen.

### 11.3.2 Version Control Systems (VCS)

- **Lokal:** SCCS (1972), RCS (1982).
- **Zentralisiert:** CVS (1986), Subversion (2000). Ein zentraler Server speichert alle Versionen; Clients checken Dateien aus.
- **Verteilt (Distributed):** Git (2005), Mercurial (2005). Jeder Client besitzt eine vollständige Kopie des Repositories.

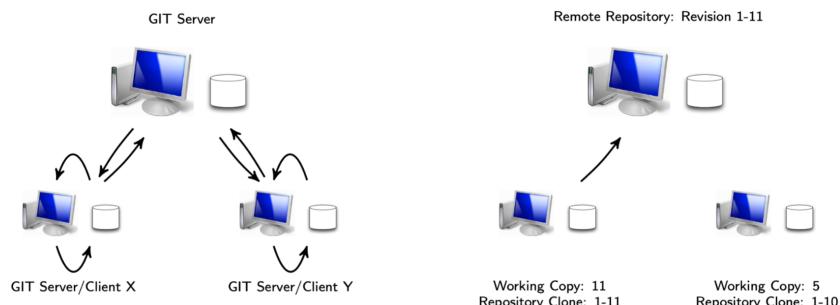


Abbildung 31: Centralized vs. Distributed Version Control

### 11.3.3 Terminologie der Versionskontrolle

- **Repository:** Eine Sammlung von Versionen einer Menge von Dateien.
- **Working Tree:** Die lokal ausgecheckte Version der Dateien.
- **Commit:** Ein Schnappschuss (Snapshot) des Zustands des Repositories zu einem bestimmten Zeitpunkt.

- **Branch:** Ein unabhängiger Entwicklungszweig.
- **Staging Area (Index):** Vorbereitungsbereich für den nächsten Commit.
- **.gitignore:** Darin enthaltene Dateipfade werden von VCS ignoriert

#### 11.3.4 Operationen von Version Control Systems

- **Clone:** Lokales Repository erstellen von einem Remote Repository
- **Fetch:** Remote Repository herunterladen
- **Commit:** Änderungen am lokalen Repository speichern
- **Push:** Lokales repository zu Remote Repository hochladen
- **Merge:** Zusammenführen zweier Branches

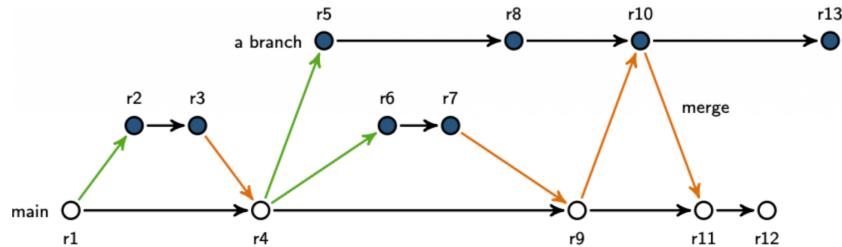


Abbildung 32: Branching and Merging

#### 11.3.5 Gute Commit-Nachrichten

- **Titel des Commits**
  - Kurze Zusammenfassung (72 Zeichen oder weniger)
  - Sollte beendet werden mit “if applied, this commit will...”
- **Nachrichten-Body (optional)**
  - Eine Zeile Abstand gefolgt vom Body
  - Erkläre was sich geändert hat und warum

## Conventional Commits

- **fix:** fixen eines Bugs (patch)
- **feat:** einführen eines neuen Features (minor change)
- **BREAKING CHANGE:** einführen einer neuen API Veränderung (major change)
- **refactor:** Refactoring (verändern) von Code
- **Semantic Versioning:** major.minor.patch

### 11.3.6 Merging

- **Automatic Merge:** VCS übernimmt automatisch Veränderungen für den Merge
- **Merge Conflict:** VCS kann nicht automatisch mergen, Benutzer muss den Konflikt selbst fixen

## 11.4 Systembau (System Building)

Der Systembau ist der Prozess der Erstellung einer ausführbaren Systemversion durch das Kombinieren von Komponenten, Konfigurationsdateien und Bibliotheken.

### 11.4.1 Build-Umgebungen

- **Development System:** Die Umgebung des Entwicklers (inkl. Compiler, Debugger).
- **Build Server:** Ein dedizierter Server für die Erstellung offizieller Versionen.
- **Target System:** Die Umgebung, in der das ausführbare System später läuft.

### 11.4.2 Werkzeuge für Build und Systemintegration

- **Build-Skript-Erstellung:** Identifikation abhängiger Komponenten, automatisierte Generierung.
- **Versionskontroll-Integration:** Auschecken benötigter Versionen.

- **Minimale Neukompilierung:** Bestimmung, welche Teile neu kompiliert werden müssen.
- **Testautomatisierung:** Ausführung automatisierter Tests (z. B. Unit-Tests).
- **Reporting und Dokumentation:** Berichte über Erfolg/Fehler und Generierung von Release-Notes.

## 11.5 Continuous Integration (CI) und Deployment (CD)

„Agile Methoden empfehlen sehr häufige System-Builds mit automatisierten Tests zur frühzeitigen Erkennung von Softwareproblemen. Häufige Builds sind Teil des Prozesses der kontinuierlichen Integration.“

### 11.5.1 Schritte in der kontinuierlichen Integration

1. Clone/Fetch aus dem Versionskontrollsysteem.
2. Lokal Build durchführen und automatisierte Tests ausführen.
3. Änderungen implementieren.
4. Lokal erneut testen; bei Erfolg: Commit in einen Feature-Branch.
5. Commit löst Build auf dem Server aus.
6. Bei erfolgreichen Server-Tests (und Code-Review): Merge in den Hauptentwicklungs Zweig (Main).

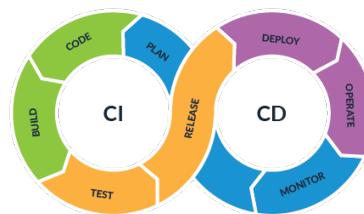


Abbildung 33: Continuos Integration & Development

## 12 Folie 12: Software Evolution

### 12.1 Lehman's Laws of Software Evolution

- Anhaltender Wandel: Systeme müssen konstant geändert werden um nützlich zu bleiben
- Zunehmende Komplexität: Komplexität nimmt zu, wenn nichts dazu beigetragen wird es zu reduzieren
- Erhaltung der Vertrautheit: Zufriedenstellende Evolution schließt übermäßiges Wachstum aus
- Anhaltendes Wachstum: Funktionalität muss kontinuierlich erweitert werden, um Zufriedenheit der Nutzer zu gewährleisten
- Abnehmende Qualität: Qualität sinkt, wenn System nicht kontinuierlich gewartet und an Änderungen der Einsatzumgebung angepasst wird

#### Konsequenz

- Benutzte Software wird verändert
- Jede Veränderung erhöht Komplexität
- Qualität sinkt ohne Gegenmaßnahmen

### 12.2 Bedeutung von Software Evolution

Je älter und schlechter gepflegt → desto teurer jede Änderung.

1. Make it correct
2. Make it clear
3. Make it concise
4. Make it fast

Performance ist letzter Schritt, nicht erster. Sauberer Code lässt sich später leichter optimieren.

### 12.3 Software Evolution Faktoren

- **Interne Faktoren:** veränderte Business Prozesse oder Praktiken
- **Technologische Faktoren:** neuere Plattformen oder Libraries
- **Externe Faktoren:** neue oder veränderte Gesetzlage

### 12.4 Arten von Veränderungen

- **Corrective Change:** Bugfixes, Designfehler beheben
- **Adaptive Change:** Anpassung an neue Plattformen, Bibliotheken, Technologien
- **Perfective Change:** Neue oder veränderte Anforderungen implementieren

### 12.5 Code Refactoring

#### Definition 12.1: Code Refactoring

Struktur verbessern, ohne Funktionalität zu ändern. Refactoring muss kontinuierlich passieren, nicht nur am Ende.

### 12.6 Code Smells

#### Definition 12.2: Code Smells

Code Smells sind Strukturen, die Wartbarkeit verschlechtern. Sie sind Warnsignale, keine Bugs.

#### 12.6.1 Refactoring Methoden

- **Pull Up Method:**
  - Eine Methode, die in mehreren Unterklassen identisch implementiert ist, wird in die gemeinsame Oberklasse verschoben.
  - Dies eliminiert Redundanz und zentralisiert die Logik (DRY-Prinzip).
- **Extract Method (Methode extrahieren):**
  - Ein Codefragment innerhalb einer zu langen oder komplexen Methode wird in eine neue, separate Methode ausgelagert.

- Die neue Methode erhält einen aussagekräftigen Namen, der beschreibt, *was* das Fragment tut, wodurch die Lesbarkeit und Wiederverwendbarkeit erhöht wird.

- **Rename Refactoring (Umbenennen):**

- Bezeichner (Variablen, Methoden, Klassen) werden umbenannt, um deren Zweck im aktuellen Kontext präziser widerzuspiegeln.
- Dies ist eine der einfachsten, aber effektivsten Maßnahmen zur Verbesserung der Code-Verständlichkeit und zur Reduktion von Fehlinterpretationen.

### 12.6.2 Beispiele

- **Mysterious Name**

- Schlechte Bezeichnungen von z.B Funktionen → Verständnisproblem
- **Lösung:** Rename Refactoring

- **Duplicated Code:**

- Gleiche Logik mehrfach vorhanden → Änderungen müssen überall angepasst werden → Inkonsistenz entsteht
- **Lösung:** Extract Method Refactoring

- **Long Method**

- Lange Funktionen schwer verständlich. Wenn man einen Kommentar braucht, schreibe stattdessen eine Funktion.
- **Lösung:** Extract Method Refactoring

### 12.7 Zusammenfassung

Software verändert sich zwangsläufig und wird dabei ohne gezielte Gegenmaßnahmen immer komplexer und qualitativ schlechter. Änderungen entstehen durch neue Anforderungen, technologische Anpassungen oder Fehlerkorrekturen und führen langfristig zu strukturellem Verfall, wenn kein kontinuierliches Refactoring betrieben wird. Code Smells wie lange Methoden, doppelte Logik oder schlechte Benennungen sind Warnsignale für sinkende Wartbarkeit. Refactoring dient dazu, die interne Struktur zu verbessern, ohne das Verhalten zu ändern, und hält das System langfristig verständlich und erweiterbar. Zentrale Botschaft: Lesbarkeit und sauberes Design sind wichtiger als frühe

Optimierung, denn nur klare und strukturierte Software bleibt entwickelbar und beherrschbar.

## 13 Folie 13: Software Maintenance

### Definition 13.1: Software Maintenance

Änderung eines Systems nach Auslieferung zur Fehlerbehebung, Verbesserung oder Anpassung.

### 13.1 Arten von Maintenance

#### 1. Adaptive Wartung

- An neue Umgebung anpassen
- **Beispiel:** Windows 8 → Windows 11 Support

#### 2. Korrektive Wartung

- Fehler beheben
- **Beispiel:** Taschenrechner berechnet falsch

#### 3. Perfektive Wartung

- Verbesserung von Performance / Struktur
- **Beispiel:** Texteditor kann große Dateien besser laden

#### 4. Präventive Wartung

- Probleme verhindern
- **Beispiel:** 2000 Problem, Leap Seconds

### 13.2 Maintenance vs. Evolution

- **Software Wartung:** Prozess der Änderung eines Systems nach Auslieferung
- **Evolution:** Entwicklung + Wartung

### 13.2.1 Wartung

- kleine Änderungen
- Ungeplant
- meistens Korrekturen
- Patch Releases (2.3.1 → 2.3.2)

### 13.2.2 Evolution

- Neue Features
- Größere Änderungen
- Geplant
- Minor/Major Releases (2.3.1 → 2.4.0 / 3.0.0)

## 13.3 Reengineering

### 13.3.1 Reverse Engineering

Analyse eines bestehenden Systems, um Struktur und Zusammenhänge zu verstehen und in eine andere Darstellung zu überführen.

- Code → Modell
- **Ziel:** Verstehen
- **Beispiel:** UML-Diagramm aus bestehendem Code generieren

### 13.3.2 Forward Engeneering

Von einem Modell zur konkreten Implementierung.

- Modell → Code
- **Ziel:** Implementierung
- **Beispiel:** Anforderungen im Quellcode implementieren

### 13.3.3 Transformative Engineering / Restructuring

Transformation innerhalb derselben Abstraktionsebene, ohne das äußere Verhalten zu ändern

- Code →(verbesserter) Code
- Qualität verbessern
- **Beispiel:** Refactorings

### 13.3.4 Reengineering

Untersuchung und Veränderung eines Systems, um es in neuer Form wiederherzustellen und neu zu implementieren  
Reengineering = Kombination aus:

- Reverse Engineering
- Forward Engineering
- Restructuring

### 13.3.5 Legacy Systems

#### Definition 13.2: Legacy System

Wertvolle Software, die über lange Zeit gewachsen ist und wird oft nicht mehr von den ursprünglichen Entwicklern betreut. Das Wissen über das System ist dabei verloren gegangen

#### Typische Eigenschaften

- größer als 100k LOC
- älter als 10 Jahre
- Entwickler nicht mehr da
- Alte Technologien
- Schlechte Dokumentation
- Hohe Wartungskosten
- Business-kritisch

## 13.4 Migration

### Definition 13.3: Migration

Ein Legacy-System wird erneuert oder ersetzt.

- Das neue System ersetzt oder modernisiert das alte
- Es muss abwärtskompatibel sein
- Es soll aktuelle funktionale und nicht-funktionale Anforderungen erfüllen
- Daten müssen oft mit migriert werden
- Die Ausfallzeit soll möglichst kurz sein

### 13.4.1 Migrationsstrategien

1. **Wrapping:** Neue Software wird um das bestehende System herum gebaut. Das alte System bleibt bestehen.
2. **Redevelopment:** Die Funktionalität wird komplett neu implementiert und ersetzt das alte System.
3. **Incremental Migration:** Das System wird schrittweise erneuert. Teile werden nacheinander ersetzt. (architekturbetrieben)
4. **Stepwise Migration:** Das System wird in aufeinanderfolgenden Phasen in ein neues überführt. (funktionsgetrieben)
5. **Big Bang Migration:** Das gesamte System wird auf einmal ersetzt.

Kombinationen der Strategien sind möglich (außer Incremental und Big Bang, da sie sich widersprechen).

## 13.5 Zusammenfassung

Der größte Teil des Software-Lebenszyklus nicht in der Entwicklung, sondern in der Wartung und Weiterentwicklung liegt. Es geht darum, bestehende Systeme zu verstehen, Fehler zu beheben, sie an neue Anforderungen anzupassen und ihre Qualität langfristig zu sichern. Außerdem wird erklärt, warum viele Systeme zu Legacy-Systemen werden, welche Probleme daraus entstehen und wie man sie durch Reengineering oder Migration schrittweise oder vollständig modernisieren kann.

# 14 Folie 14: Compilation and Static Analysis

## 14.1 Grundlagen der Kompilierung

### 14.1.1 Programmiersprachen

Viele Programmiersprachen

- **Vorteile:** Für jeden Anwendungsbereich gibt's etwas
- **Nachteile:** Entwickler müssen trainiert werden für neue und tote Sprachen, Tools kosten viel Money

### 14.1.2 Compilation vs. Interpretation

- **Compilation**

- Ganzer Quellcode direkt in Zielcode (Maschinen oder Byte) übersetzen, als ausführbare Datei
- Compiler kriegt Quellcode und gibt Zielcode zrk
- **Beispiele:** C, C++, Rust oder Java, Kotlin

- **Interpretation**

- Programm wird während es läuft gelesen und übersetzt
- Interpreter kriegt Quellcode mit Eingabe Daten für Ausgabedaten
- **Beispiele:** Python, Perl, JVM

### 14.1.3 Intermediate Languages

Dienen als Brücke um Code in verschiedene Zielsprachen zu übersetzen, für Betriebssysteme wie Windows, Linux, MacOS (z.B. Java Bytecode)

- **Direkte Kompilierung:** für jede Zielsprache wird jeweils ein Compiler benötigt ( $n \times m$ )
- **Intermediate Language:** System teilt sich in Front-Ends (Quellcode zu IL) und Back-Ends (IL zu Zielcode) auf ( $n + m$ )

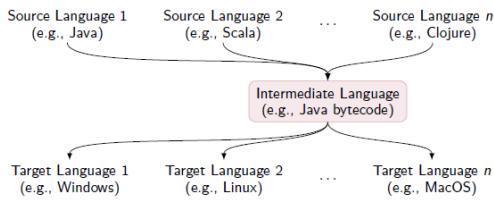


Abbildung 34: Intermediate Language

#### 14.1.4 Compiler Optimierung

- Ziel:

- Schnelles ausführen von Programmen
- Wenig Speicher-/ Energieverbrauch
- Sowohl für Compile- als auch für Runtime wichtig

- Just-in-Time (JIT) Compilation:

- Ausgeführter code wird oft während runtime compiliert
- Wenn neuer code ausgeführt wird, oft anfangs langsamer (warm-up time), bis der code compiled wurde

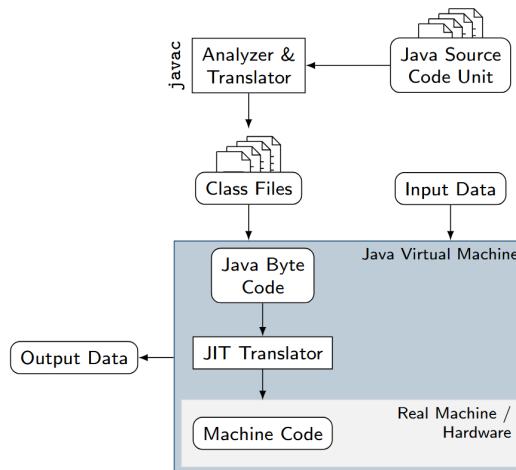


Abbildung 35: Just in Time Compiler

#### 14.1.5 Nicht-korrektter Code

- **Warnungen:**

- Zeigen potentielle Probleme oder das die Kompilierung in Zukunft fehlschlagen könnte
- Veraltete Methoden

- **Errors:**

- Zeigt das die Compilierung fehlgeschlagen ist
- Target code ist unvollständig

- **Fehlertypen:**

- Lexical errors, parke error, Type error, runtime errors, logical errors

#### 14.1.6 Compiler Architecture

Anwendung der Chomsky Hierarchy

- **Scannen/Lexing:** Verwandelt reguläre Sprachen und wandelt den Quellcode in einen Token Stream um
- **Parsen:** Nutzt kontextfreie Grammatiken (eine Klasse hat Felder, Methoden und innere Klassen) und erzeugt aus Token einen Syntaxbaum
- **Namen-/Typen Analyse:** Kontextsensitive Analyse wird ausgeführt und Abstract Syntax Tree (AST) wird generiert

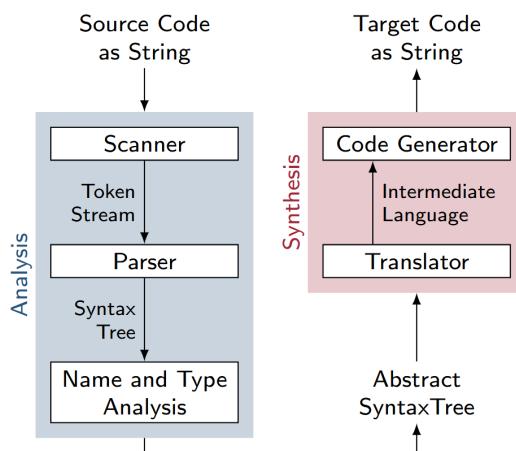


Abbildung 36: Compiler Architecture

#### 14.1.7 Typensicherheit und Korrektheit

- **Typensicherheit:**

- Ein Typ charakterisiert Eigenschaften von Programmelementen, zum Beispiel:
  - \* Eine Variable kann nur bestimmte Werte speichern
  - \* Ein Ausdruck gibt nur bestimmte Werte zurück
  - \* Ein Objekt hat eine Methode mit bestimmter Signatur
- **Type Errors** passieren, wenn Eigenschaften nicht erfüllt sind
- Ein Programm ist **Type Safe**, wenn das Ausführen nicht zu solchen Fehlern führen kann

- **Type Errors Beispiele:**

- Zuweisung eines inkompatiblen Typs
- Methodenaufruf mit inkompatiblem Parameter

- **Typen-Korrektheit:**

- Die Sprachenspezifikationen definieren Typregeln, kontrolliert vom Compiler (**Statically Typed Language**) oder Interpreter (**Dynamically Typed Language**)
- Die Gesamtheit aller Typregeln ist das **Typensystem**
- Programm ist **typkorrekt**, wenn Typregeln erfüllt sind
- Programmiersprachen sind **strongly typed** (stark typisiert), wenn alle typkorrekten Programme typsicher sind
- Ansonsten **weakly typed**

- **In der Praxis:** Kontinuum zwischen stark (Java) und schwach (JavaScript) typisierten Sprachen

#### 14.1.8 Klassifizierung von Fehlerarten

- **Lexikalische Fehler:** Treten während der lexikalischen Analyse auf, wenn Zeichenfolgen keinem gültigen Token der Sprache zugeordnet werden können. Beispiel: Ungültige Symbole oder falsch formatierte Bezeichner.
- **Syntaxfehler:** Werden vom Parser erkannt, wenn die Anordnung der Token gegen die formale Grammatik der Programmiersprache verstößt. Beispiel: Fehlende Semikolons oder nicht geschlossene Klammern.

- **Typfehler:** Teil der semantischen Analyse; hierbei sind Operationen syntaktisch korrekt, aber auf unzulässige Datentypen angewendet. Beispiel: Addition eines Strings zu einem Integer.
- **Laufzeitfehler:** Fehler, die erst während der Programmausführung auftreten, obwohl der Code syntaktisch und semantisch korrekt kompiliert wurde. Beispiel: Division durch Null oder Zugriff auf einen Null-Pointer.
- **Logische Fehler:** Das Programm läuft ohne Absturz durch, liefert aber aufgrund eines fehlerhaften Algorithmus ein falsches Ergebnis. Diese Fehler werden meist durch Techniken wie das **Software Testing** oder **Code Reviews** aufgedeckt.

## 14.2 Missverständnisse bezüglich Performance

### 14.2.1 Compiler-Optimierungen

- Arten der Optimierungen:
  - **Maschinenabhängige:** Ausnutzen von Eigenschaften einer bestimmten Maschine
  - **Maschinenunabhängige:** Auf mehreren Maschinen anwendbar
  - **Lokale:** z. B. Reihenfolge von Anweisungen ändern
  - **Intraprozedurale:** Betreffen nur eine Methode
  - **Interprozedurale:** Betreffen mehrere Methoden oder benötigen globales Wissen

### 14.2.2 Missverständnis 1: Arrayzugriffe

**Behauptung:** Arrayzugriffe sind langsam, deshalb versuche ich sie zu umgehen.

- Aufgaben für Arrayzugriff  $a[b]$ :
  - Auswerten vom Ausdruck  $b$
  - Berechne **Offset** =  $b * \text{Größe jedes Feldes}$
  - Berechne **Speicheradresse** = Position von  $a + \text{Offset}$
  - Greife auf die Speicheradresse zu
  - Nur für Objekte: Verwende den Wert als Speicheradresse

- **Anmerkung (Realität):** Arrays kommen extrem häufig vor, weshalb Compiler und Hardware massiv darauf optimiert sind.
- **Fazit:** Nur Gelaber.

### Example

```
a[b.n()] = a[b.n()-1] * a[b.n()-1];
```

### Simplified Example

Assumption 1: method n has no side-effects

Assumption 2: method n cannot be overridden  
(e.g., a private method)

```
int n = b.n();
a[n] = a[n-1] * a[n-1];
```

Abbildung 37: Array Misconception

### 14.2.3 Missverständnis 2: Schleifen

**Behauptung:** Schleifen sind langsam, deshalb versuche ich sie zu umgehen.

- **Aufgaben für Schleifen:** Erstelle/Initialisiere Variable, Bedingung prüfen, Körper ausführen, Inkrementieren, Wiederholung.
- **Schleifen nicht umgehen:**
  - Manuelles Umgehen erweckt den Anschein von doppeltem Code und langen Methoden
  - **Compiler-Optimierung: Schleifenentrollung** (Loop Unrolling), wenn die Anzahl der Durchläufe statisch bekannt und klein genug ist.

### Example Loop

```
for (int i = 0; i++; i < 3) { a[i] = i; }
```

### Loop Unrolling

```
a[0] = 0; a[1] = 1; a[2] = 2;
```

Abbildung 38: Loop Misconception

#### 14.2.4 Missverständnis 3: Methodenaufrufe

**Behauptung:** Methodenaufrufe sind langsam, deshalb versuche ich sie zu umgehen.

- **Aufgaben für jeden Aufruf:** Parameter/Rücksprungadresse übergeben, Register speichern, Rumpf ausführen, Rückgabewert übergeben, Register wiederherstellen.
- **Methodenaufrufe nicht umgehen:**
  - Vermeidung führt zu dupliziertem Code und schlechtem Design
  - **Refactoring-Ziel:** Methoden extrahieren für bessere Lesbarkeit
  - **Compiler-Optimierung: Methoden-Inlining** (Inhalte werden direkt in den Aufrufer kopiert)

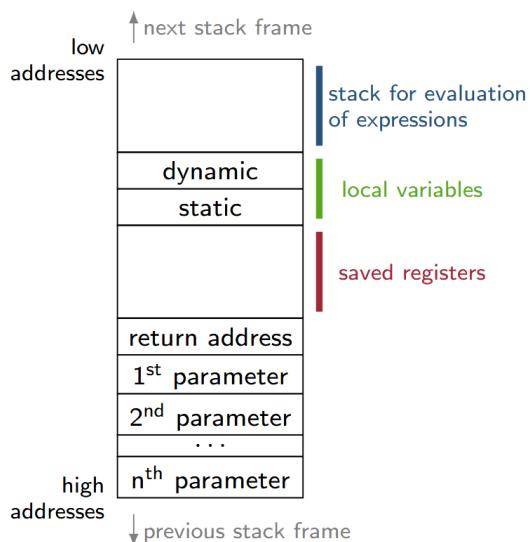


Abbildung 39: Method Call Misconception

#### 14.2.5 Missverständnis 4: Objekte

**Behauptung:** Objekte sind langsam, deshalb versuche ich sie zu umgehen.

- **Aufgaben für Objekte:** Heap-Speicherung (wenn Lebensdauer lang), Zeiger/Dereferenzierung, Klasseninfos speichern, Virtuelle Methodentabelle (Dynamic Dispatching).
- **Objekte vielleicht nicht vermeiden:** Sofern Leistung oder Speicherverbrauch kein Problem darstellt. Wichtig ist die Identifizierung der Objekte, die den meisten Speicher belegen bei kürzester Lebensdauer.

#### 14.2.6 Missverständnis 5: Garbage Collection

**Behauptung:** Garbage Collection (GC) sei langsam, deshalb versuche ich sie zu umgehen.

- **Garbage Collection:** Sucht nach nicht-referenzierten Objekten und gibt Speicher frei (Referenzzählung, Mark-and-Sweep).
- **GC nicht vermeiden:**
  - Vereinfacht Programmieren
  - Weniger Speicherlecks
  - Verbesserte Sicherheit und Zuverlässigkeit

### 14.3 Test-Driven Entwicklung und Design per Vertrag

#### 14.3.1 Fehlerarten

- **Typfehler:** Werden vom Compiler vor dem Lauf erkannt (z. B. falsche Argumente)
- **Laufzeitfehler:** Erst zur Laufzeit erkannt (z. B. `NullPointerException`)
- **Logische Fehler:** Programm läuft durch, liefert aber falsches Ergebnis
- **JUnit / TDD:** Zuerst Test schreiben, dann Code implementieren, Assertions nutzen.

#### 14.3.2 Design by Contract (Methodenverträge)

- **Motivation:** Verhindert unübersichtlichen Code durch „defensives Programmieren“. Informale Spezifikationen veralten oft.
- **Konzept:** Formale Spezifikation direkt im Code.
  - **Vorbedingungen / Nachbedingungen** für Methoden
  - **Klasseninvarianten** für den Zustand der Klasse
- **Beispiel: Java Modeling Language (JML)** für Dokumentation, Laufzeitzusicherungen und formale Verifikation (KeY).

### 14.3.3 Generated Assertions und Blame Assignment

Bei Vertragsbruch hilft das System, den Schuldigen zu finden:

- **Problem bei Vorbedingung:** Geprüft beim Caller → **Caller schuld** (ungültige Werte übergeben).
- **Problem bei Nachbedingung:** Geprüft beim Callee → **Callee schuld** (falsches Ergebnis berechnet).

### 14.3.4 Behavioral Subtyping

- Beschreibt Verhalten von Verträgen bei der **Vererbung**.
- Eine Unterklassie muss sich an die Verträge der Elternklassie halten.
- **Liskovsches Substitutionsprinzip:** Programm muss beim Ersetzen eines Objekts der Oberklassie durch eine Unterklassie weiterhin korrekt funktionieren.

## 15 Folie 15: Software Product Lines

### 15.1 Software Product Lines and Feature Modeling

#### 15.1.1 Grundlagen

- **Mass Customization:** Bezieht sich auf Massenproduktion von kundenindividuellen Produkten mit dem Ziel, diese so effizient wie möglich in Masse zu produzieren.
- **Softwareproduktlinie (SPL):**
  - Satz an softwareintensiven Systemen (Produkten), die sich einen gemeinsamen, verwalteten Funktionsumfang teilen und Anforderungen erfüllen.
  - Erstellt aus einem gemeinsamen Satz an Assets (Reuse).
- **Feature:** Eine Domänenabstraktion, die der Kommunikation zwischen Stakeholdern dient und Unterschiede zwischen Produkten spezifiziert.

### 15.1.2 Feature Modeling

- **Feature Model:** Stellt eine Hierarchie von Features dar.
- **Abhängigkeiten:** Werden durch einen Baum oder baumübergreifende Einschränkungen modelliert.
- **Tree Constraints** (Baumeinschränkungen): Durch die Hierarchie definiert.
  - Jedes Feature benötigt sein Elternteil.
  - **Optional:** Ein optionales Feature hat keine weiteren Beschränkungen.
  - **Mandatory** (Obligatorisch): Feature ist für das Elternteil gefordert.
  - **Or group:** Mindestens ein Feature muss ausgewählt werden, wenn das Parent-Feature ausgewählt wurde.
  - **Alternative group:** Genau ein Feature muss ausgewählt werden.
- **Cross-tree Constraints:** Baumübergreifende Einschränkungen in Form von aussagelogischen Formeln über Features.
- **Feature-Typen:**
  - **Konkrete Features:** Haben eine Implementierung.
  - **Abstrakte Features:** Dienen zum Eingruppieren anderer Features.

### Vorteile

- **Darstellung von Variabilität:** dienen dazu, alle verfügbaren Merkmale (Features) eines Systems und deren Beziehungen zueinander abzubilden.
- **Configuration Management:** gültige Kombinationen von Software für eine spezifische Zielumgebung
- **Effizientes System Building:** automatisieren einen Teil des System Buildings
- **Beherrschung der Komplexität:** helfen die "Increasing Complexity" (Lehman's Law) zu bewältigen

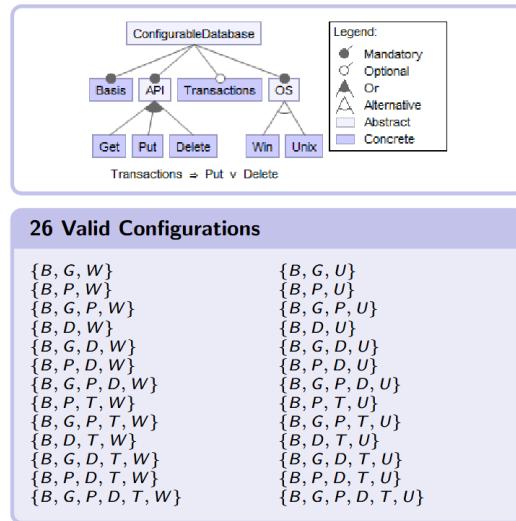


Abbildung 40: Feature Model

### 15.1.3 Valid Configurations

Das Endziel des Feature Modelings ist es, festzulegen, welche konkreten Produkte gebaut werden können. Jede Kombination von Features, die alle Tree- und Cross-Tree-Constraints erfüllt, stellt eine gültige Konfiguration dar.

## 15.2 Implementierung von Software Product Lines

### 15.2.1 Strategien

- **Branching & Merging:** Gleichzeitige, unabhängige Entwicklung mit der Option zum Mergen in der Zukunft.
- **Laufzeitparameter (Runtime Parameters & Properties):**
  - Gesamter Code ist im Kompilat enthalten, wird aber dynamisch ein- oder ausgeschaltet.
  - Features können beim Start via Kommandozeile oder über einen internen Property-Manager (z. B. aus einer Konfigurationsdatei) abgefragt werden.
- **Präprozessoren (Bedingte Kompilierung):**
  - Nutzung von Befehlen wie `#define`, `#if` und `#endif` (klassisch in C).

- Für Sprachen wie Java gibt es Tools wie **Munge** oder **Antenna**, die spezielle Kommentare nutzen (`/*if [Feature]*/`), um Code zu aktivieren oder auszukommentieren.
- **Frameworks mit Plug-Ins:** Modularste Form (Black-Box-Framework). Einzelne Features werden als eigenständige Plug-Ins entwickelt, die ein gemeinsames Interface implementieren. Ein Plugin-Loader lädt nur die konfigurierten Features.

## 15.3 Testen von Software Product Lines

### 15.3.1 Test-Ansätze

- **Testen aller Konfigurationen:** Nur bei kleinen Produktlinien praktikabel. Erzeugt enorm redundanten Testaufwand; bei großen SPLs ist oft nicht einmal bekannt, wie viele gültige Konfigurationen es gibt.
- **Testen einer einzigen Konfiguration:** Auf große Produktlinien anwendbar ohne redundanten Aufwand. Aber: Meist nicht möglich, alle Features in einer Konfiguration zu testen.
- **Gefahr:** Übersehen von **Feature-Interaktionen**.

### 15.3.2 Feature Interaktionen

Features funktionieren gut in Isolation, aber fehlerhaft in Kombination (Beispiel: Brandbekämpfung vs. Hochwasserschutz).

- **Pairwise Interactions:** Interaktion von 2 Features.
- **T-Wise Interactions:** Interaktionen von  $t$  Features.

### 15.3.3 Pairwise Interaction Testing (PIT)

Testen einer Stichprobe, die so berechnet wird, dass jedes mögliche Paar von Features in mindestens einer Konfiguration vorkommt.

- **Vorteile:** Reduziert den redundanten Aufwand enorm, auf große SPLs anwendbar und bietet garantierte Abdeckung.
- **Nachteil:** Rechnerisch sehr aufwendig, die kleinstmögliche Stichprobmenge zu berechnen.

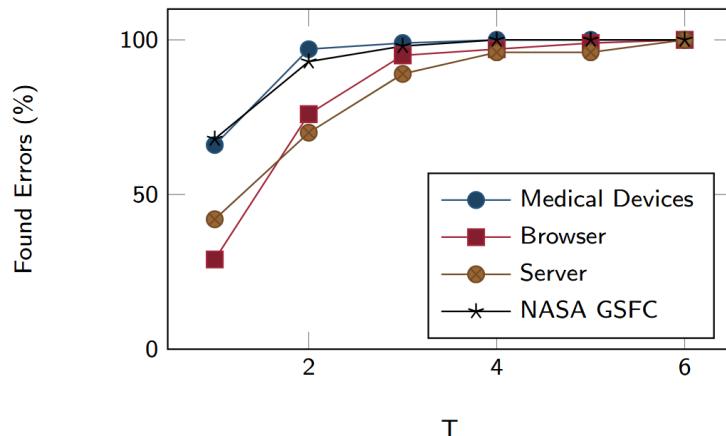


Abbildung 41: Effectivity of Interaction Testing

#### 15.3.4 Zusammenfassung und Trade-Off

- **Großes t:** Hohe Abdeckung (effektiver).
- **Kleines t:** Geringer Testaufwand (effizienter).
- **Praxisbeispiel Linux:** Linux Kernel v2.6.28.6 hat über 6.800 Features. Eine Pairwise-Stichprobe (ca. 480 Konfigurationen) zu berechnen dauert etwa 9 Stunden.

## 16 Folie 16: Open-Source Software

### 16.1 Open-Source Entwicklung

#### Definition 16.1: Open-Source Entwicklung

Source-Code wird veröffentlicht, freiwillige Entwickler können am Entwicklungsprozess teilnehmen

- Freiwillige Personen nehmen am Entwicklungsprozess teil, indem sie Bug Reports schreiben, neue Features und Änderungen anfragen
- Änderungen am Main-Repository werden von Inhabern kontrolliert

#### 16.1.1 Kriterien für Verwendung von open-source Komponenten

- Entsprechen diese Qualitätsanforderungen?

- Werden diese aktiv maintained?
- Lassen sich diese einfach integrieren?
- Ist es günstiger / einfacher diese weiterzuentwickeln?
- Wie sind diese lizenziert?

#### 16.1.2 Richard Stallman's Four Freedoms

1. Freiheit es für egal welchen Verwendungszweck auszuführen
2. Freiheit den Source-Code einzusehen und das Programm zu modifizieren
3. Freiheit das Programm weiterzuverteilen
4. Freiheit Freiheit es weiterzuverteilen oder sogar modifizierte Versionen zu verkaufen

#### 16.1.3 Free Software

- **Free Software** (Freie Software): Quellcode ist offen und zugänglich. Jeder darf die Software untersuchen, verändern und verbreiten.
- **Freeware**: Software, die kostenlos verwendet werden darf, deren Rechte allerdings komplett beim Entwickler bleiben.
- GNU: Free Software Foundation

### 16.2 Open-Source Principles

- Open-Source Software gehört einem Unternehmen oder Individuum
- Lizenzen bestimmen Einschränkungen über die Verwendung der Software
- Ist keine Lizenz angegeben, darf der Code nicht verwendet werden, modifiziert oder verteilt werden
-

### 16.2.1 Copyleft und Copyright

#### Definition 16.2: Copyleft

- Recht, Eigentum zu modifizieren und weiterzugeben
- Eine veränderte Version muss unter derselben Lizenz stehen
  - **Weak Copyleft:** trifft nur auf zuvor lizenzierte Teile zu
  - **Strong Copyleft:** trifft auf die komplette Software zu
- **Beispiel:** GNU General Public License (GPL) für Software, Creative Commons für Dokumente und Bilder

#### Definition 16.3: Copyright (Urheberrecht)

- Geistiges Eigentum, Patente, Marken
- Eigentümer hat exklusives Recht
- Niemand darf ohne Erlaubnis kopieren, verändern oder verbreiten.
- **Ziel:** Urheber wird wirtschaftlich geschützt, wenn jemand die Software verwenden will, muss dieser eine Lizenz kaufen oder um Erlaubnis fragen

## 16.3 Open-Source Lizenzen

### 16.3.1 MIT License

- Vom Massachusetts Institute of Technology 1987 veröffentlicht
- Erlaubt Wiederverwendung proprietärer (nicht quelloffene) Software
- Lizenz wird mit dem Vertrieb ausgeliefert oder weiterlizenziert (auch für proprietäre Lizenzen)
- "provided as-is, without warranty"
- **Permissiv:** sehr weitreichende Freiheiten und nur minimale Einschränkungen (im Gegensatz zu Copyleft)

### 16.3.2 GPL: GNU General Public License

- Software, die (modifizierte) GPL Software verwendet muss ebenfalls unter GPL veröffentlicht werden (Strong Copyleft)
- Für jede Veröffentlichung oder Weiterverteilung muss der Source Code verfügbar sein
- Für privaten oder interne Verwendung: Keine Verpflichtungen
- Für kommerzielle Zwecke erlaubt

### 16.3.3 LGPL: GNU Lesser General Public License

- Software, die GPL Software verwendet muss nicht unter LGPL veröffentlicht werden
- Modifizierungen von LGPL Software müssen allerdings unter LGPL veröffentlicht werden

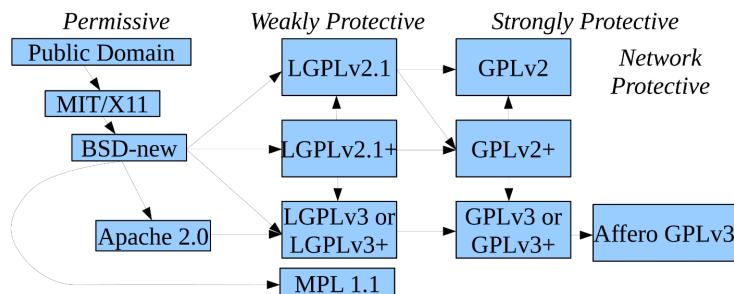


Abbildung 42: Open Source License Overview

### 16.3.4 Re-Licensing und Dual-Licensing

Generell sind Lizenzen miteinander inkompatibel

- **Re-Licensing**: Software muss entfernt werden, neu implementiert und eine neue Lizenz vergeben werden
- **Dual Licensing**: Eigentümer ist nicht an die eigene Lizenz gebunden, kann mehrere Lizenzen haben

# 17 Folie 17: Automotive Software Architectures

## Definition 17.1: AUTOSAR

AUTomotive Open System ARchitecture

- Referenz Architektur + Entwicklungs-Methodik + AUTOSAR Plattform
- **AUTOSAR Classic Platform:** für traditionelle mechatronische Systeme z.B. Türen, Klimaanlage
- **AUTOSAR Adaptive Platform:** für moderne Systeme z.B. autonomes Fahren, Konnektivität

## 17.1 Layered Architecture

Referenz-Architektur für Autonomes Fahren

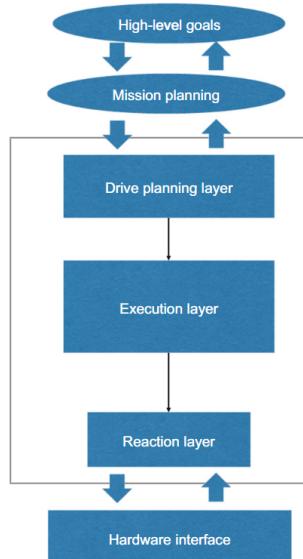


Abbildung 43: Automotive Layered Architecture

## 17.2 Pipe-and-Filter Architecture

Pipe-and-Filter Architecture für Bildverarbeitung

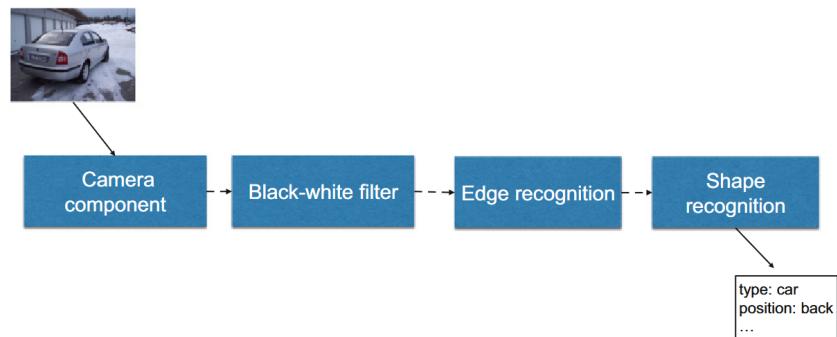


Abbildung 44: Automotive Pipe And Filter Architecture

### 17.3 Client-Server Architecture

Client-Server Architektur des Flottenmanagement

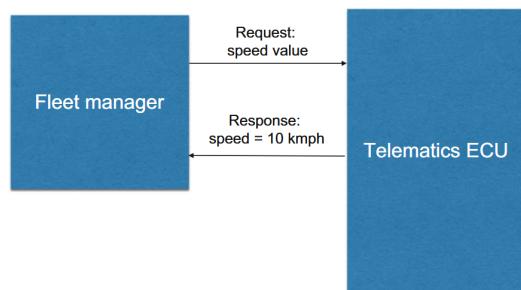


Abbildung 45: Automotive Client Server Architecture

### 17.4 Publish-Subscribe Architecture

Ähnlich dem Observer pattern aber auf Architektur-Ebene

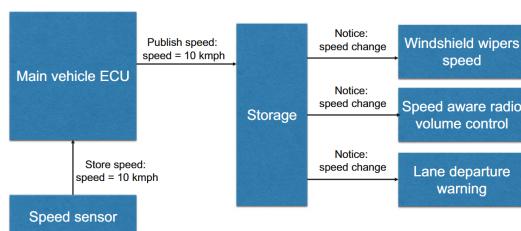


Abbildung 46: Automotive Publish Subscriber Architecture

## 17.5 Centralized Software Architecture

Redundanz erforderlich als fail safe

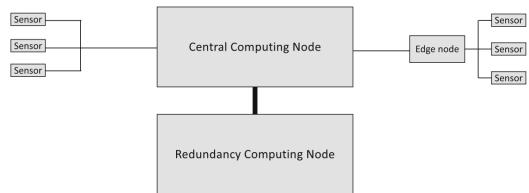


Abbildung 47: Automotive Centralized Software Architecture